Simulation of Autonomous Robot Teams with Adaptable Levels of Abstraction

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Vom Fachbereich Informatik der
Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
genehmigte

Dissertation

von

Dipl.-Inform. Martin Friedmann
(geboren in Groß-Gerau)

Referent:	Prof. Dr. rer. nat. Oskar von Stryk
Koreferent:	Prof. Dr. Eng. Enrico Pagello (Universität Padua, Italien)
Tag der Einreichung:	19.10.2009
Tag der mündlichen Prüfung:	30.11.2009

D17
Darmstadt 2010

Für meine Frau, die mir den Mut gegeben hat, diese Arbeit zu beginnen, und die Kraft, sie zu Ende zu führen.

Acknowledgements

This thesis was created during my time as research assistant at the *Simulation, Systems Optimization and Robotics Group (SIM)* at the *Department of Computer Science* of the *Technische Universität Darmstadt*.

I wish to express my gratitude to my supervisor Prof. Dr. rer. nat. Oskar von Stryk. Without him I never would have started working in the field of humanoid robots. During many discussions he helped me to shape this thesis. When I was in need of advice, he never was more than an email away (and often just one door).

Likewise I want to thank Prof. Dr. Eng. Enrico Pagello for becoming second referee of my thesis. I am grateful for his interest in my work and for his supportive comments.

Thanks go to all my current and former colleagues at the SIM group. They have contributed to my thesis in many ways: sharing thoughts and coffee, demanding new features from my simulations, or simply being there when a second pairs of eyes was needed when debugging.

I also wish to thank all members of the *Darmstadt Dribblers* RoboCup Team. Many of them have been beta testers for parts of the software I developed for this thesis. Likewise I wish to thank the members of the Research Training Group 1362 *Cooperative, Adaptive and Responsive Monitoring in Mixed Mode Environments* for stimulating discussions, especially in the context of search-and-rescue-robots.

Most of all I wish to thank my family. Without their constant support and patience, I would not have been able to finish this thesis.

Contents

List of Figures

List of Tables

Symbols

μ	friction constant
ρ	gear ratio of a servo-motor
τ	force/moment acting in a joint
ω	angular velocity of an object
$\dot{\omega}$	angular velocity of an object
d_{col}	penetration depth of a collision
$\mathbf{e_r}$	axis of a revolute joint (represented as a unit vector)
$\mathbf{e_t}$	direction of a prismatic joint (represented as a unit vector)
$\mathbf{f_{ext}}$	external force acting on an object
I	inertia tensor of an object
J	inertia of a motor's rotor
m	mass of an object
$\mathbf{n_{ext}}$	external torque acting on an object
q	position of a joint
$\mathbf{n}_{col}$	normal of a collision
$\mathbf{p}_{col}$	position of a collision
$\dot{q}$	velocity of a joint
$\ddot{q}$	acceleration of a joint
$\mathbf{r}$	position of an object
R	orientation of an object (represented as a rotation matrix)
$\mathbf{v}$	linear velocity of an object
$\dot{\mathbf{v}}$	linear acceleration of an object

1 Introduction

Autonomous robots nowadays are used for many complex tasks. Among the most prominent applications in recent years are soccer playing robots, e. g. in the international RoboCup competitions, or autonomous cars, e. g. in the challenges held by the DARPA.

Developing and programming of autonomous robots is a complex and therefore error-prone task. To achieve a dependable and capable system, a large variety of hard- and software components must be developed and tested. Testing of these components is required on different levels [117], like single modules, e. g for sensor data interpretation, localization, behavior control or motor control, interaction of such modules of one robot or interaction between robots of a team. One of the most valuable tools used for these tasks of development and programming is the simulation of robots.

A robot simulation is a computer program which has the ability of simulating the robot's sensors, its motion apparatus and its interaction with the environment. In this thesis *simulation* is used synonymously for *robot simulation*, if it is not clear from the context, that something else is being simulated.

A robot simulation consists of two main components which have to be specified in concert: The *model* describing the simulated robot(s) and their environment and a set of *computational simulation methods* capable of calculating the simulated aspects of the robot (e. g. motion, sensing or interaction). Which kind of information is represented in the model highly depends on the computational simulation methods used in a concrete simulation. Depending on the level of abstraction the method is using, different information must be provided by the model. Parts of this thesis deal with the problem of modeling robots for simulations in a way that allows multiple simulation methods providing different levels of abstraction to be used with the same model.

In the context of software development for autonomous robots, many applications exist for simulations. The most straightforward application obviously is using a simulation as a replacement of the real robot, thus enabling tests of the complete control software of the robot under certain conditions and restrictions.

As the simulation can access ground truth information about the simulated robot and its environment, it is able to provide input on the robot's and the environment's state on different levels of abstraction which cannot be observed by the robot's real sensors. Thus it is possible to bypass parts of the robot's control software like sensor processing or world-modeling and directly provide proper information on the environment to normally hidden parts of the control software. This allows for an isolated testing of these modules without the danger of masking errors of the module under consideration by problems introduced at other levels.

Likewise it may be of interest to simulate the motion of an autonomous robot on another than the physical level. By choosing proper levels of abstraction often the amount of CPU time required for the simulation can be reduced dramatically, thus enabling the simultaneous simulation of multiple robots. Besides the obvious tradeoff between accuracy of the simulation and the number of robots simulated, other advantages could be drawn from a simulation with several levels of detail and abstraction. One example for this is to use a simulation method which does *not* simulate certain disturbances of the robot's motion. By using a simulation with

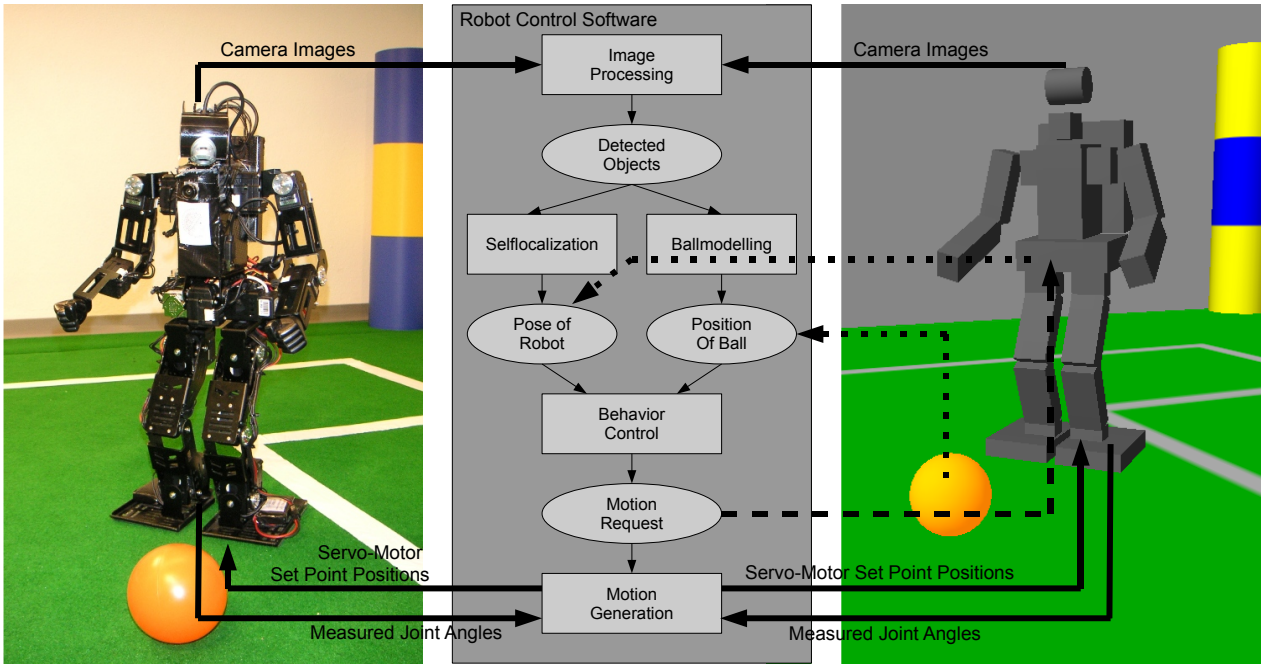


Figure 1.1: A simulation of a humanoid robot providing multiple levels of abstraction. The simulation is able to process motion commands of the control software and to provide sensor information on the same level of abstraction as the real robot does (solid arrows). Further sensor information can be provided on a higher level of abstraction, skipping several modules of the control software (dotted arrows). Likewise motion generation can be skipped by sending more abstract high level motion commands directly to the simulation (dashed arrow). Image adapted from [62]

this reduced level of detail it can be determined, if an observed erroneous behavior of the robot has been caused by an error in the control software or by an external disturbance.

An example for a simulation of a humanoid robot providing several levels of abstraction is given in Figure 1.1.

Unlike real hardware, which is often expensive and of limited availability, a simulation can be used repeatedly and simultaneously by many developers, thus rising productivity. Further on, investigation of teams of robots is often only possible using simulations as not enough real robots are available. An impressive example for this are the RoboCup competitions: While soccer matches with real hardware still are limited to few robots (depending on the league 3 to 5) per team, in the 2D simulation league matches are played with 11 agents per team.

Another application area for simulations is hardware development of robots. In this area simulations can be used to test a design even before producing a prototype, thus saving time and cost in case of faulty or weak designs.

Besides applications in the field of robot development, simulations are of great importance in education. A simulation allows for easy experimentation in the field of robotics without endangering real hardware. Often it is not possible to provide enough real robots for classroom purposes. In this case simulations provide a feasible way for teaching robotics to even large audiences. Again it is of interest to provide the simulation of a robot system on different levels of abstraction. For a quick introduction abstract models of the robot can be used, which later

on could be refined to provide more complex tasks to the students, e. g. by adding errors to the sensor or by simulating more realistic (and thus less stable) motions like legged locomotion.

Due to these benefits, many different simulations for autonomous robots have been developed in recent years. Depending on the simulation's purpose, these simulations greatly differ in what kind of robot they can simulate, which aspects of a robot are simulated and how accurate each of the aspects is simulated. Most of the current robot simulation programs for autonomous robots provide specific aspects at exactly one level of abstraction, e. g. *only* 2D kinematical motion simulation or *only* 3D dynamical simulation. If a robot is to be simulated at different levels of abstraction, often the only solution is using multiple simulation programs. This approach though, requires to model the robot multiple times and to interface multiple simulations with the robot's control software, thus multiplying efforts as well as sources of errors when setting up simulation experiments.

1.1 Contents and Contribution

The aim of this work is to develop a new methodology which allows the simulation of single as well as of teams of autonomous robots with different levels of abstraction in a degree of flexibility which has not been possible before. Simulations based on this methodology shall be able to provide simulation methods for motion and sensor simulation of autonomous robots in a way that allows an easy combination of several of such methods for simultaneous use.

Chapter 2 gives an overview of the state of research relevant to this thesis.

In Chapter 3 methodological considerations for the simulation of teams of autonomous robots on different levels of abstraction are made. Initially the requirements on a simulation depending on typical use cases are investigated leading to a classification of abstraction levels for motion- and sensor simulation. After this a methodology is derived which allows for a structured analysis and validation of simulation methods on different levels of abstraction and accuracy. By applying this validation methodology the user is enabled to make an informed decision on which simulation methods to use for a specific application. The chapter closes with a set of design considerations and requirements for simulations with an adaptable level of abstraction.

As part of the research done for this thesis, the Multi Robot Simulation Framework (`MuRoSimF`) has been developed following the design considerations specified in Chapter 3. The following three chapters discuss details of this framework and which considerations were made to support an adaptable level of abstraction.

In Chapter 4 a flexible modeling approach is derived allowing to describe a wide variety of different robots. To support an adaptable level of abstraction, the modeling approach focuses on an easy exchange of the simulation methods connected to the modeling data.

Several methods for the simulation of robot motion and sensing have been implemented as part of `MuRoSimF`. An overview of these methods and their different levels of abstraction is given in Chapter 5.

The integration of models and simulation methods into an exexutable simulation is discussed in Chapter 6.

As part of this work, several simulations for autonomous robots have been developed and applied successfully for the development of software for biped, quadruped and wheeled robots. Each of the simulations makes use of the techniques developed in order to provide the simulation on a level of abstraction adequate to the different purposes of the simulations. An overview of these simulations is given in Chapter 7.

The thesis closes with a summary and and conclusions in Chapter 8.

The main contribution of this work is the development of a new methodology for robot simulations which provide multiple levels of abstraction. This methodology covers aspects of robot modeling, implementation of simulation methods and validation of simulations. Based on this methodology the Multi Robot Simulation Framework (`MuRoSimF`) has been developed. This framework enables the easy recombination of computational simulation methods providing different levels of abstraction. Several simulations for different kinds of robots and different application have been developed successfully using `MuRoSimF`, demonstrating the usability and versatility of the methodology.

2 State of Research

2.1 Robot Control Software

In this section a brief overview of central concepts and techniques used in robot control software is given. Normally several different modules exist within a robot's control software. In Section 3.1 of this thesis it will be discussed, which simulation methods are appropriate for testing specific modules.

2.1.1 Overview

Control takes place on different levels of an autonomous robotic system as described in [29]. High level control tasks are used to decide on the robot's next actions, usually based on the input of external sensors, e. g. planning a path through an environment or deciding on the robot's role in a soccer game. Low level control tasks are used to execute these actions on the robot, e. g. by generating stable motion trajectories for a legged robot.

Concerning the way a robot processes sensor input, different strategies have been investigated. Several different approaches can be used to decide which actions the robot should take next depending on the input from its sensors as described in [101]. Deliberative systems are based on planning the robot's action based on an internal representation of the current state of the world (the world-model) and subsequently executing these actions. These techniques are highly dependent on the accuracy of the world model and cannot react quickly to changes in the environment. Reactive systems (e. g. [36]) do not rely on a world-model but directly react to input of the sensors, thus allowing immediate reaction to changes in a dynamic environment. As these techniques do not use a world-model, their use is limited to applications which do not require long term planning or learning. Hybrid systems combine the advantages of deliberative systems (e. g. strategic planing) with those of reactive systems (e. g. fast response to changes in the environment).

Even though these strategies differ substantially in how they solve the main tasks, the general structure of a robot's control software boils down to three areas:

- **Sense** uses the robot's sensors to provide information on the state of the robot and its environment.
- **Plan** uses the information provided by Sense to decide the robots next actions, e. g. moving around or manipulating the environment
- **Act** executes the actions decided by plan on the robot.

Sense, plan and act may appear on different levels within a control architecture, e. g. on servo-motor level (sense provides information on one motor's position and velocity, plan is the motor's control strategy and act switches the motor's control voltage), on motion planning level or on whole robot level.

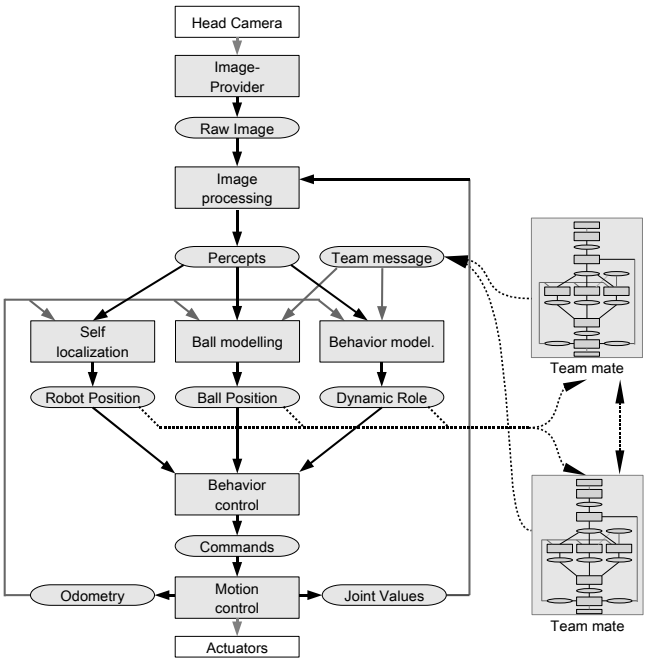


Figure 2.1: Example for the software architecture of an autonomous humanoid robot [117]. Modules performing calculations are depicted as boxes, exchanged data as ellipses.

2.1.2 Components of Robot Control Software

Typically the control software for an autonomous robot consists of several modules to accomplish the tasks of the sense-plan-act cycle. Depending on the robot's field of application for each of these tasks, different approaches may be chosen and combined to the specific robot's control architecture. Figure 2.1 shows an example for a typical control architecture.

The following sections give a brief overview of widely used methods for the different tasks appearing a robot's control software.

Sensor Processing

Depending on the subsequently used techniques, sensor processing provides different informations. In case of an unknown environment this usually are detected features within the sensor's reading (e. g. detected corners in the scan of a laser-range-finder or features like SIFT [99, 100] or SURF [26] in a camera image). If the environment (or at least the objects therein) is known, also well defined objects may be detected (e. g. an orange ball in case of a robot-soccer setting). Detected features and objects initially are represented within the respective sensor's coordinate frame. For subsequent processing of these informations, it may be necessary to transform this information to other coordinate frames, e. g. a common frame for all robots or even world coordinates (if the robot's pose within the environment is known).

World Modeling

The world modeling module or modules are used to provide a model of the robot's environment as well as of the robot's situation within this environment. Depending on the robot's task world modeling may handle one or more different tasks like self-localization, mapping or modeling of mobile objects in the environment. The world-model is usually fused from information provided from several internal (like odometry or inertial sensors) and external sensors (like cameras, bumpers or laser range finders). In teams of robots the robots may maintain a common world model or use information provided by other robots to update their own model [128].

The task of a robot's self-localization is to provide information on the robot's position and orientation (the combination of both will further on be called *pose*) within its environment. If the robot's pose can be determined uniquely from the information present (e. g. if triangulation is always possible), a single pose can be provided. If this is not possible, often a probability distribution is provided. A widely used technique providing an unimodal estimation is the Kalman-filter [81]. If a unimodal representation is not sufficient, e. g. in case of ambiguous information, other techniques must be used. Typical methods are banks of Kalman-filters, representing the probability distribution as a discretized grid (e. g. [55]) or particle filters which represent the probability distribution as a set of particles (e. g. [142]).

If no map of the environment is known a-priori, a map must be built by the robot and simultaneously the robot must localize itself in this map. This problem is referred to as SLAM (simultaneous-localization-and-mapping), an overview is given in [47, 48].

Beyond the robot's pose and the map, the world model may contain further information on the environment. These may be the positions of mobile obstacles (e. g. [140]), other robots in a team (e. g. [128]) or other mobile objects of interest (e. g. the ball in a robot-soccer match [92]).

A comprehensive overview of probabilistic techniques used for self-localization of robots with and without mapping is given in [141].

Behavior Control

The task of a robot's behavior control is to make a decision on the robots next action(s). These decisions may be based on the robot's world model, on direct sensor input or on combinations thereof. A wide range of methods have been developed for this purpose, e. g. the classical subsumption architecture [36], hierarchical finite state machines [121, 98], problem solving based on logic programming [97, 73] or behavior descriptions based on petri-nets [153]. A comprehensive overview on recent techniques can be found in [120].

Motion Planning and Generation

Motion planning and generation for autonomous mobile robots is handled on two distinct levels.

High level planning is concerned with the question of finding a path for the robot within its environment. Depending on the scenario, this plan can be based on different information which may be known from the start (like a map) or which may be provided by the robot's sensors (like - possibly mobile - obstacles). A wide variety of techniques has been investigated for this problem, an overview can be found in [93].

Low level planning is concerned with control of the robot's joints resp. motors in order to perform the desired motion. Depending on the kind of robot different approaches may be applied. For wheeled robots the velocity of the wheels and the direction of the steering have to be chosen and controlled. For legged robots more complicated techniques are used, as the motions of many degrees of freedom have to be coordinated. Prominent approaches to this problem are central pattern generators deriving all joint positions as a function directly depending on a central clock (e. g. [28]) or motion planning techniques for the robot's center of gravity and feet deriving joint angles from inverse kinematics (e. g. [56]). To maintain the stability of a legged robot, additional techniques based on criteria for static stability [102] or dynamic stability (e. g. the zero-moment-point [145] or the foot-rotation-indicator [70]) are often applied to guide the motion generation. Sometimes motion generation for legged robots has a further intermediate level for planning a sequence of steps, e. g. [91]. The motion generation for legged robots often depends on a large set of parameters (like swing height or step frequency). To find optimal gaits for a robot, often optimization techniques are applied (e. g. [75, 109, 50]).

2.2 Robot Motion Simulation

To simulate the motion of an autonomous mobile robot, several tasks must be solved. The robot's motion itself must be simulated using an appropriate algorithm. Depending on the required accuracy of the simulation, this can be done by a purely kinematical simulation or, if higher levels of accuracy are required, by simulating the robot's dynamics. In case of dynamics simulation or if interaction with the environment is to be considered by the simulation, collisions between the robot and the environment (and in some cases, collisions of the robot with itself) must be detected and handled.

2.2.1 Algorithms for Multi-Body-Systems

Multi-Body-Systems (MBS) are systems of rigid bodies connected by rotational and prismatic joints. A rigid body is described by its dimensions, the positions of the joints, its mass, center of mass and inertia tensor.

MBS Kinematics.

The position vector $\mathbf{r} \in \mathbb{R}$ and orientation (represented as a rotation matrix) $R \in \mathbb{R}^{3\times3}$ of each body of an MBS can be calculated recursively from the base, depending on the position and orientation of the base as well as the positions q of the joints. In the same way it is possible to calculate velocities and accelerations. These calculations are called *direct kinematics* and can be accomplished by an efficient ($O(n)$ runtime with n being the number of joints of the structure under consideration) algorithm (e. g. [45]).

Calculating the joint positions required for a given position and orientation of a body of an MBS is called *inverse kinematics*. Solving this problem is often non-trivial and is highly dependent on the kinematical structure of the MBS. Often the solution is not unique. Algebraic and geometric methods for deriving the robot specific equations of inverse kinematics are discussed in [45].

MBS-Dynamics.

The dynamics of the unconstrained motion of a single rigid body are described by

$$\mathbf{F} = m\ddot{\mathbf{r}} \tag{2.1}$$
$$\mathbf{N} = I \cdot \dot{\omega} + \omega \times (I \cdot \omega) \tag{2.2}$$

with **F** and **N** being the force and moment vectors attacking at the center of mass of the body and I being the inertia tensor of the body at the center of mass[45].

The coupling of several rigid bodies within a multi-body-system leads to constrained motions of the individual bodies. Several approaches have been developed to handle the resulting systems of equations. These approaches can be classified into algorithms using generalized (joint-angle) coordinates and algorithms using 3D coordinates of the bodies.

MBS-Dynamics in Generalized Coordinates

Consideration of the relation between the acceleration of the generalized coordinates $\ddot{\mathbf{q}} \in \mathbb{R}^n$ of the joints of an MBS and the forces and moments τ acting within theses joints leads to the field of MBS dynamics. For an open loop structure of rigid bodies the dynamics can be described by the equation

$$\tau = M(\mathbf{q}) \cdot \ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}}) + \mathbf{G}(\mathbf{q}).$$

In this equation the mass matrix $M \in \mathbb{R}^{n \times n}$ describes the relation of accelerations and forces of the joints, $\mathbf{C} \in \mathbb{R}^n$ are the Coriolis forces resulting from the motion of the system and $\mathbf{G} \in \mathbb{R}^n$ are forces caused by gravity. Additional consideration of external forces (e. g. caused by contact) leads to

$$\tau = M(\mathbf{q}) \cdot \ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}}) + \mathbf{G}(\mathbf{q}) + \mathbf{F}(\mathbf{q}, \dot{\mathbf{q}})$$

with $\mathbf{F} \in \mathbb{R}^n$ being the impact of these forces on the individual joints.

Calculation of the joint forces and moments τ needed to yield a desired acceleration $\ddot{\mathbf{q}}$ is known as calculation of the inverse dynamics. A well known algorithms for calculation of the inverse dynamics is the Recursive Newton Euler Algorithm (RNEA) [53]. The RNEA is an efficient algorithm with an $O(n)$ runtime (with n being the number of joints. It does not require explicit calculation of the mass matrix M.

When simulating the dynamics of a robot, it is necessary to calculate the accelerations of the robot's joints resulting from the forces acting in these joints. The resulting problem is called the calculation of the robot's *forward dynamics*. The naive approach to solving this problem is an explicit calculation of the mass matrix followed by solving the resulting set of linear equations. One well known algorithm following this approach is the Composite Rigid Body Algorithm (CRBA) [146] which is based on a column wise calculation of the mass matrix using the RNEA. Even though the calculation of the mass matrix has an $O(n^2)$ complexity, the algorithms complexity is $O(n^3)$ due to the ensuing solving of the equations. More efficient algorithms avoid the explicit calculation of the mass matrix. One well known algorithm allowing calculating the forward dynamics with $O(n)$ complexity is the Articulated Body Algorithm (ABA) [51]. In [52] it is shown that the CRBA is faster than the ABA for $n \leq 8$.

Drives

Besides the kinematic and dynamic properties of an MBS the properties of the drives have a large impact on the equations of motion of the whole system. Depending on the level of detail used in modeling the robot as well as the simulation methods used, different effects may be considered. Several methods have been established which allow the integration of effects of drives into the generalized coordinates formulation of MBS dynamics.

If only the kinematics of the motion are of interest, the drives will be modeled as devices defining the position q, velocity $\dot{q}$ or the acceleration $\ddot{q}$ of a joint. When simulating the dynamics of motion, the model of the drive must provide information on the torque (resp. force) τ exerted on the joint by the drive. Simple approaches will model the robot's drives as a method of introducing a torque to the joint, sometimes with additional constraints modeling some properties of the drive (e. g. maximum torque and velocity in [138]).

More complex models will consider further properties of the drive, e. g. caused by the motor or the gears. A widely used approach in robotics (e. g. [52]) is to model the rotor inertia of the motor as well as the gears as an additional inertia acting only in the respective joint (thus neglecting gyroscopic effects). In an inverse dynamics model an additional torque is calculated for each joint to overcome this inertia. For a gear-ratio ρ and an rotor-inertia J this torque is

$$\tau_r = \rho^2 \cdot J \cdot \ddot{q}.$$

Even though the rotor inertia is usually small compared with the inertia of the rigid bodies of a robot, the above term may have significant influence on the dynamics of motion, if a high gear ratio is used (as it is often the case in servo-motors).

Another effect commonly modeled is friction in the joint. A multitude of models differing in level of detail have been presented for this purpose (for an overview see [111]). Widely used effects considered in robotics are:

- Viscous friction. A friction which linearly depends on the velocity of the joint causing a torque $\tau_f = \mu_v \cdot \dot{q}$ with the friction constant μ_v (e. g. [65, 43, 17]).
- Coulomb friction. A constant friction $\tau_f = \mu_c \operatorname{sign}(\dot{q})$ (e. g. [17]) only depending on the direction of the motion. In [43] a similar model is used with different values for the friction constant μ_c depending on the sign of $\dot{q}$.

Other effects may be considered depending on the respective application. Often several effects are combined into a model, as in [43, 17].

Advanced methods exist to consider further effects caused by the drives like elasticity of the gears or gyroscopic effects of the rotor (both e. g. [77]).

MBS-Dynamics in 3D Coordinates

Several classes of methods considering the motion of systems of rigid bodies in 3D coordinated have been developed. These methods differ in the way constraints caused by links between the bodies (e. g. joints) or contacts are resolved. Widely used methods are:

- *Penalty* methods (e.g. [107]) are based on calculating the motion for each body individually as if it were not connected to any other body in the system. This inevitably leads to breaking constraints defined between the bodies. To restore a broken constraint, additional penalty forces (e. g. by springs and dampers) are introduced in order to move the bodies so that the constraint holds again.

- Methods using *Lagrange multipliers* have been widely used in animation (e. g. [25, 24]): These methods are based on setting up a (usually sparse) linear system of equations to calculate the forces required to keep the constraints. Contacts lead to additional inequality constraints which can be handled as a linear complementary problem (LCP) [22]. Generally methods based on Lagrange multipliers have $O(n^3)$ runtime. In [23] an $O(n)$ method for solving a set of loop free constraints is presented.
- In [31, 32] a method based on *impulses* has been described. This method iteratively calculates small impulses which are applied to the single bodies of the MBS in order to satisfy all constraints. Calculation of the impulses is based on a solving a linear system of equations. A linear-time solution for acyclic models is given in [30].

2.2.2 Dynamics Packages

Several software packages providing functionalities for the calculation of a robot's dynamics are available. Often these packages are not limited to the calculation of the direct dynamics, but provide further functionalities like integration or collision-detection and -handling. Theses packages may be used as the core component of robot simulations. In the following sections several packages widely used in robot simulations are presented and discussed.

Open Dynamics Engine

The Open Dynamics Engine (ODE) [135, 134] is an open-source package providing simulation for rigid body dynamics as well as collision detection. The rigid body dynamics simulation is based on the Lagrange-multiplier method (see Section 2.2.1). Two distinct solvers are provided: an accurate solver with $O(n^3)$ runtime and and approximate iterative solver with $O(m \cdot n)$ runtime with m being the number of iterations and n the number of constraints. Motors can be simulated by setting the position, the velocity or the force for a joint, but there there is no support for simulating the joint's drive. Collision detection is provided for primitive shapes as well as for meshes using the GIMPACT library [9].

NVIDIA PhysX

PhysX [7] is a package aimed at the computer games industry. Besides other features it provides rigid body and vehicle physics as well as collision detection. PhysX allows multithreaded execution on CPUs, NVIDIA graphics hardware and special acceleration boards. The package is available for Windows and Linux platforms as well as some game consoles. A SDK is freely available, but the project is not open source, thus preventing inspection of the algorithms used.

Bullet

Bullet [10, 44] is an open-source library for collision detection and physics simulation which is used in several computer games and animation packages (including Blender). Rigid body dynamics simulation is based on the Lagrange-multiplier method, collision detection is provided for primitive shapes and meshes based on GIMPACT[9]. Further features of Bullet include

dynamics for soft and deformable objects as well as for vehicles. Bullet can be executed multithreaded and has been optimized for several CPUs and GPUs. The library has been ported to many different systems including game consoles.

Unreal Engine / Karma

The Unreal Engine [8] is a commercial game engine produced by Epic Games which has been used in the "Unreal" computer games series and several other games. Of special interest in this discussion is version 2.5, as it has been used as the base for the USARSim simulation (see Section 2.3). Among other features it provides scripting, high quality rendering and a rigid body physics simulation. The rigid body simulation is based on the Karma package which is based on a Lagrange-multiplier method. Besides the dynamics solver, Karma provides collision detection covering primitive shapes and meshes.

Newton Game Dynamics

Newton Game Dynamics [6] is a library providing collision detection and rigid body physics. It is provided as a binary release for Windows, Linux and Mac OS X, but no sourcecode is publicly available. The current version 1.53 has been available since 2006, a new version featuring multi-processor support has been announced, but has not yet left beta stadium.

Discussion

The packages described above all provide rigid-body physics (most with several kinds of joints) and collision detection, thus allowing basic simulation of the motion dynamics of a robot. None of them, though, seem to consider more robot specific features (like the dynamics of a joint's drive). The simulation methods used by the packages are often documented poorly.

2.2.3 MBS Simulation Software

For engineering applications a wide range of tools for the simulation of mechanical systems has been developed. Typical tools of this group are MD Adams[1], SimPack[2] or SimMechanics[3]. Due to the commercial character of these systems, few information on their functioning, especially on the modeling and simulation methods used, is publicly available. The focus of such tools lies on the development and simulation of mechanical systems and related control strategies, thus their use in the context of the simulation of autonomous mobile robots is limited, as they are missing crucial components like the simulation of external sensors.

[1] http://www.mscsoftware.com/Contents/Products/CAE-Tools/MD-Adams.aspx
[2] http://www.simpack.com/439.html
[3] http://www.mathworks.de/products/simmechanics/

2.3 Simulations for Autonomous Mobile Robots

Many simulations for autonomous mobile systems exist. Typically these simulation combine the simulation of a robot's (or a team of robots) motion with some simulation for the sensors used. The simulations can be classified by several criteria.

One central criterium is the kind of robots which can be simulated. Some simulators only can simulate one specific robot or a specialized class of robots (like robot's with one specific mode of locomotion). Other simulators are more general and allow the simulation of robots with different modes of locomotion and a wide range of sensors.

Further criteria are the kind of simulation (e. g. kinematical or dynamical motion simulation, simulation in 2D or 3D), the number of robots which can be simulated simultaneously, the kind of modeling used for the robots and the availability of the simulation (available platforms and licensing).

In the following section, an overview of existing simulations which are currently widely used is given. Afterwards the features of these simulations are discussed. The section closes with a brief discussion of the topic of validation of simulations.

2.3.1 General Simulations

In this section simulations are discussed which are capable of simulating a wide variety of robots.

Blender for Robotics

Blender[4] is a very advanced open-source 3D modeling and animation package. Besides high quality rendering, it provides a wide variety of technologies useful for animations including calculation of inverse kinematics, dynamics simulation (based on the Bullet library) and scripting (based on Python) as well as an integrated game engine.

It has been suggested to use the technologies provided by Blender to create a simulation for mobile robots [39]. As documented on [1, 2] some progress was made towards the creation of the so called *OpenRobots Simulator*. A simulation of a wheeled vehicle equipped with a camera has been presented which could by controlled using the YARP[5] libraries. Integration with other robotics frameworks is also considered.

Microsoft Robotics Studio

Microsoft Robotics Studio [5, 79, 108] (MSRS) provides a simulation module. The simulation is integrated in the MSRS and thus uses the service-oriented communication mechanism provided by MSRS.

Physics simulation for robots and environment is based on the PhysX API. Besides the physical simulation several sensors including bumpers, laser range finders and cameras can be simulated. The properties of all simulated entities can be accessed through services. By implementing additional services, new sensors can be simulated based on these properties.

As reported in [104], a humanoid robot has successfully been simulated using MSRS. In this simulation several different joints with differing parameters were used. The servo-motors driv-

[4] http://www.blender.org

[5] Yet Another Robot Platform – http://eris.liralab.it/yarpdoc/index.html

ing these joints could be added easily by adding new services to the simulation. It is reported that the simulation model could reproduce different kinds of motion including walking.

Simbad

Simbad [78, 13] is a multi robot simulation developed in Java. Motion simulation is constricted to the 2D plane and based on a kinematic motion model with simplified physics handling collisions of objects and robots. Sensors simulation is provided in 3D for cameras, distance sensors and contact sensors. Visualisation and camera simulation is based on 3D rendering using the Java3D-API, distance sensors are simulated using the selection methods provided by Java3D. The environment and the simulated robots are described by Java-classes which can be extended by the user. Likewise the robot control software is provided by Java classes.

SimRobot

SimRobot [96, 95] is a general 3D robot simulation which has been used for wheeled [96], quadruped [125] and biped robots [95]. The robot's physics is simulated using ODE with extensions to support PID controllers to simulate servo-motors. The camera simulation is based on OpenGL rendering. It is capable of reproducing distortion effects caused by motion of the camera on a high level of accuracy [113]. Distance sensors may be simulated by the computation of ray-object-intersections based on ODE (for single ray sensors only) or by reading the OpenGL depth buffer. Touch sensors are simulated using collision information provided by the physics engine. The simulated robots are described using the XML based language `RoSiML` [68]. Integration of the simulation with the respective control software is done by linking. The core of the simulation runs single threaded. The software is freely available as part of the code release of the B-Human RoboCup-team [124].

Stage and Gazebo

Stage and Gazebo are two multi-robot simulations which where developed in combination with the Player robot device server [67, 42].

The Stage Simulation [66] provides a simplified 2D simulation for mobile robots situated in a world described as a bitmap. Mobile robots are modeled as a mobile base with additional sensors using a description language. The mobile base can move in any direction of the 2D plane and rotate around its upward axis. The motion can be controlled in several ways, e. g. simulating a differential drive, an omnidirectional drive or a car like control. Several models of sensors are provided, including 2D laser range finders and sonars, bumpers, abstract cameras providing information on color segmented blobs and abstract sensors providing information on the relative position of specific objects. The simplified models allow the simulation of large groups of robots. Recently Stage has been updated to use 3D representations of the robots and the environment [144]. While motion simulation remains limited to the 2D plane, visualisation, camera simulation and the user interface now are 3D.

The simulation can be accessed in several ways by the robot control software. When running the simulation standalone, the control software can be integrated as libraries with the simulation. Alternatively the simulation itself can be integrated as a library into other software projects. Using this technique, the simulation can cooperate with a Player-server allowing the robot-control software to communicate with the simulation using TCP connections.

The Gazebo simulation [89] is a 3D simulation. Motion simulation is based on ODE. Several different sensors including monocular and stereo cameras, laser rangefinders, sonar, GPS and

inertial sensors can be simulated. The camera simulation is based on OpenGL rendering, the simulation of distance sensors is based on the calculation of ray object intersections provided by ODE. In [112] an extension to Gazebo allowing the simulation of omnidirectional cameras based on ray casting in combination with OpenGL-rendering is presented. A simulation of omnidirectional cameras based on cube mapping is presented in [27]. Additionally In this paper approaches for multi level abstraction for the simulation of sensors are discussed.

Gazebo provides direct integration with the robot control software as well as use of the Player server.

When using the Player server, a robot control program can be connected transparently to either of the two simulations or a real robot.

USARsim

USARsim [40, 147], the "Unified System for Autonomous Robot Simulation" (formerly called "Urban Search And Rescue Simulation") is a 3D robot simulation initially developed for vehicles performing search and rescue operations in an urban scenario. The simulation is based on the Unreal-Engine 2.0 which is used in the commercial computer game Unreal-Tournament 2004. To access the properties of the game engine, USARSim makes use of GameBots [16, 82], which provides a network interface to the game. By using a commercial game engine, the simulation is able to use the engine's physics simulation (based on the Karma-Engine [3]) and graphics capabilities. The simulation itself is realized as a set of scripts in the game's scripting language as well as game levels for the scenarios.

Beyond its initially intended use, the simulation has been used successfully for the simulation of humanoid robots [71, 72], quadruped robots [152] and marine vessels [40]. Besides the simulation of the robot's motion capabilities USARsim provides a wide range of external sensors including distance sensor (single and multiple rays), cameras (including omnidirectional camera simulation [127]), GPS (considering motion of the satellites [21]) as well as specialized sensors (e. g. for victims in a search and rescue scenario or for the level of sound). Internal sensor simulation includes an odometry sensor (providing a pose estimate based on dead reckoning of the wheels) and joint encoders. The simulation has been validated by comparing the performance of real and simulated sensors [41, 20] as well as the motion of real and simulated robots [114].

While the simulation has been widely used, several problems arise from its dependency on a commercial game engine. The game engine can not be accessed directly but only through the scripting interface of the game. USARsim provides a wide variety of modeling options and parameters for modeling scenes and robots.Nevertheless not all parameters may be chosen independently. An example for these limitations is given in [104], where it is reported that the parameters for describing servo-motors and joints could not be chosen independently for different joints within the same simulation.

The game engine only allows limited access to data generated by the rendering subsystem. To access the images created (which is necessary for the simulation of cameras), an auxiliary program has to be used to capture images created by the engine. As the engine is used for games with first person perspective, special measures have to be taken to simulate multiple cameras at the same time. When doing so, all cameras have to share one screen for rendering, thus reducing the maximum resolution of the cameras. As it is not possible to extract the depth buffer information created during rendering, this information can not be used for the rapid simulation of distance sensors.

Further limitations are imposed by the underlying game engine's physics simulation. In [104] it is reported that it is not possible to speed up or slow down simulated time and that the physical precision of the simulation is reduced to keep a good frame rate. It should be noted that these limitations are perfectly well made tradeoffs for a computer game, but not necessarily optimal for a simulation.

A simulation based on USARsim can be accessed via a TCP/IP connection by the control application. Several interfaces for accessing the robots are provided, including Player [66] and MOAST [130].

While USARsim itself is freely available, the underlying game, Unreal-Tournament 2004, has to be bought.

Webots

Webots [106, 14] is a commercial 3D robot simulation system system. General motion simulation is based on rigid body physics provided by ODE, but a specialized kinematic algorithm for differential drives is provided as well. The simulated robot's joints can be controlled by a servo-motor-model allowing P-control of the motor's position as well as setting the force of the joint directly. Additionally, elasticities may be simulated by a spring and damper model associated with the servo-motor. No efforts are made to simulate further effects of the motor, e. g. rotor inertia. Several kinds of sensor including cameras, distance sensors and inertial sensors can be simulated. Simulation of cameras is based on OpenGL rendering of the scene. Gaussian noise can be added to the simulated camera images, but no further effects like lens distortion or motion blur are available. By reading the OpenGL depth buffer, the camera simulation also can be used to simulate range finders. Other distance sensors are simulated by calculations of ray intersections with the scene. Simulated sensor values optionally can be mapped to arbitrary output characteristics including noise. The simulated scene including the robots is described using VRML[6] with additional nodes to define robot specific information (e. g. mass, inertia and friction).

Webots provides programming interfaces for several programming languages including C, C++, Java, Python and Matlab. For each simulated robot a so called controller can be started by the simulation. Controllers may be executed synchronized with the simulation or asynchronously. To provide a connection to external robot control software, a controller can initialize a TCP/IP based network connection.

The simulation algorithms provided by Webots can be extended to some extent by plugins. Interfaces exist to interact with the ODE simulation (allowing to introduce special external forces), to add custom 2D simulation algorithms for robots with differential drive and to add sound propagation algorithms.

2.3.2 Specialized Simulations

Besides the general purpose simulations presented in the previous section a wide variety of specialized simulations for autonomous robots has been developed. In this section a selection of the most prominent ones is presented.

[6] The Virtual Reality Modeling Language, cf. http://www.web3d.org/x3d/specifications/#vrml97.

OpenHRP

OpenHRP [83, 11] is a platform for software development and simulation for robots. It provides a simulation for the simulation of a robot's dynamics as well as sensors. The simulation consists of a set of CORBA servers, one for each core module of the simulation (dynamics simulation, collision detection, visualisation, robot controller and model loader), thus enabling distributed computation as well as transparent replacement of each of the modules. This feature is made use of by providing two distinct implementations for the dynamics simulation. OpenHRP is available freely from the project's website[7].

RoboCup 3D soccer simulation

For the use in the 3D soccer simulation league of RoboCup the SimSpark simulation has been used with different robot models in recent years[33]. This simulation is based on the Spark framework [110], providing physics simulation based on ODE, dynamic management of the simulation's objects, scripting of the simulation and network communication with the software agents controlling the simulated robots. Starting with abstract spheres representing the robots the simulation has evolved to a simulation of humanoid robots. Currently the Nao robots also used in the Standard-Platform league are simulated in 5 vs. 5 games. While a physical simulation of the motion is provided, the simulation does not simulate cameras or other real-world external sensors. Instead an omnidirectional object sensor is used [34]. The simulation is developed in a joint effort of teams participating in the 3D simulation league. It is available freely and open source[8].

UCHILSIM

UCHILSIM [149] is a simulation for quadruped robots (Sony AIBO ERS-220 and ERS-7). It provides dynamic simulation of the robot's motion based on ODE. Simulation of the robot's camera is based on OpenGL rendering. Additional postprocessing of the images with Gaussian blur, noise, radial lens distortion and aberrations is applied to simulate effects of the camera's CMOS sensor [151] using techniques presented in [19]. Further on the robot's joint position sensors, acceleration sensors and contact sensors are simulated. The simulation provides an application interface to allow integration with OPEN-R software [151]. Using an iterative process named *Back to Reality* of performing experiments on real and simulated robots alike, the simulation has been used to improve robot motions and behaviors while at the same time optimizing the parameters of the simulation model [148, 150]. Development of the simulation seems to have stopped. The last update on the project's website[9] is from late 2004. Only a binary demonstration version of the simulation is available for download.

Übersim

Übersim [38] is a framework dedicated to the physical simulation of robots. Physics simulation is based on ODE, visualization and camera simulation on OSG[10] and OpenGL. It has been reported to be used for the simulation of wheeled robots in the context of the RoboCup-SmallSize-League [38] as well as for a balancing two wheeled Segway RMP robot [69]. The communication between the simulation (server) and the control software (client) is based on

[7] http://www.openrtp.jp/openhrp3/en/about.html
[8] http://simspark.sourceforge.net/
[9] http://www.robocup.cl/uchilsim/
[10] The OpenSceneGraph, cf. http://www.openscenegraph.org/projects/osg

TCP. Special measures were taken to ensure synchronized execution of client and server including simulation of latency in the communication. A version of the simulation's sourcecode from 2004 is freely available on the project's homepage[11].

2.3.3 Summary

In this section the main features of the discussed simulations concerning the simulation of motion and external sensors are summarized. Further on the availability of different levels of abstraction is discussed.

Motion Simulation

Most of the presented simulations provide exactly one method for the simulation of the robot's motion. In case of most of the 3D simulations, this is based on 3rd party tools for dynamics simulation like ODE, PhysX or Karma. The only exceptions from this are OpenHRP which provides two interchangeable implementations for the dynamics simulation and Webots which allows the additional use of a 2D kinematics simulation in combination with the 3D dynamics simulation. Two simulations are limited to 2D motion: Stage uses a kinematical simulation and Simbad a dynamical simulation. An overview of the motion simulation provided by the discussed simulation programs is given in Table 2.1.

Extendability

Only OpenHRP and Webots provide well defined interfaces for exchanging the motion simulation method. In OpenHRP it is possible to use other implementations for the dynamics simulation and the collision detection through a CORBA interface. Webots provides plugin-interfaces to load extensions to the 3D dynamics simulation (but not to exchange the ODE core) and to load 2D kinematics simulation.

Abstraction

With the exception of OpenHRP and Webots none of the discussed simulations provides a way for using different levels of abstraction for the motion simulation within the same simulation. Even these are limited: Webots only allows the exchange of simulation methods (and thus abstraction) for robots with differential drive and OpenHRP only allows the complete exchange of the dynamics simulation, but no combination of different levels of abstraction. Using the combination of Stage and Gazebo it is possible to simulate the same robot in a 2D or a 3D environment providing the same interface, thus providing different levels of abstraction by exchange of the whole simulation.

Simulation of External Sensors

This section summarizes the capabilities for the simulation of external sensors found in the discussed simulations. An overview is given in Table 2.2.

[11] http://www.cs.cmu.edu/ robosoccer/ubersim/

Table 2.1: Comparison of used methods for motion simulation.

Simulation	Type	based on	Extendability	Remarks
Blender	3D dyn.	Bullet		
Gazebo	3D dyn.	ODE		
MSRS	3D dyn.	PhysX		
OpenHRP	3D dyn.		CORBA interfaces for collision detection and dynamics simulation	Two distinct implementations are provided.
Simbad	2D dyn.			
SimRobot	3D dyn.	ODE		
SimSpark	3D dyn.	ODE		
Stage	2D kin.			
Übersim	3D dyn.	ODE		
UCHILSIM	3D dyn.	ODE		
USARsim	3D dyn.	Karma		
Webots	3D dyn.	ODE	plugin interface to ODE core	3D and 2D simulation can be used together
	2D kin.		plugin interface to add arbitrary simulation methods for robots with differential drive	

Cameras

Most of the presented 3D simulations (with the exception of SimSpark) provide simulation of cameras based on 3D rendering. Only few simulators though, go beyond pure rendering using the standard perspective projection model for the camera. Effects caused by the camera's motion and the rolling shutter are only addressed in SimRobot. Radial distortion caused by standard lenses is addressed in Uchilsim and [19]. Simulation of omnidirectional cameras is reported for USARsim and Gazebo. For both simulations a method based on a cube mapping the scene onto a mirror shaped surface has been implemented [127, 27]. For Gazebo additionally a method based on ray casting in combination with OpenGL-rendering has been implemented [112].

Distance Sensors

In the discussed simulations two approaches can be found for the simulation of distance sensors: ray casting (calculation of collision between rays and objects of the scene) and reading the depth buffer of the computer's rendering hardware. As shown in [60] the use of either method can be advantageous: ray casting is faster for small numbers of rays and allows arbitrary distributions while reading the depth buffer is usually better for many rays, but imposes limitations on the distribution of the rays. Several simulations provide both methods, but often the distribution for the rays is limited for the ray-casting method.

Abstraction

Most of the discussed simulations focus on simulating external sensors on device level, trying to reproduce data from a real sensor. Depending on the implementation, sometimes efforts are taken to reproduce typical errors of these sensors thus providing a higher level of realism. More abstract representations of the readings of a sensor are only found in few simulations, e. g. in

the abstract camera simulations of SimSpark or Gazebo and Stage, which provide information on objects seen by a specific sensor. If sufficient information on objects and sensors is provided, abstract cameras can be provided within the control software. In [27], a simulation of wheeled robots based on Gazebo is presented which allows to choose several abstraction levels for the sensor simulation.

Discussion

As shown in the previous section a wide variety of simulations for autonomous mobile robots are available. The discussed simulations provide a wide range of different methods for simulating the robot's motion and sensing capabilities on different levels of abstraction.

Little work has been done to support multiple levels of abstraction for the motion simulation within one simulation. Only two of the discussed simulations provide well defined interfaces to exchange the simulation methods used for motion simulation. From these, only one simulation allows the use of two different methods at the same time (but only allows the exchange of the 2D simulation for a special class of robot). All other simulations are limited to one specific simulation method.

More options exist for the simulation of sensors on different levels of abstraction. Many simulations provide special abstract sensors (e. g. for objects or for the robot's pose) which can be used instead of simulated real sensors. If the simulations provide access to ground truth data for robots and other objects, such sensors also can be implemented by the user (in case of an open source project within the simulation or for a closed source simulations within the control application).

For the open-source simulations discussed, an obvious way to provide more levels of abstraction, is changing the source code of the simulation. As most of the simulations have been developed around some existing package for physics simulation (in fact, mostly ODE), changing the motion simulation usually is very complicated, as this will inflict a major part of the simulation. Often the implementation of the sensor simulation directly accesses parts of the physics simulation (e. g. distance sensors in SimRobot and Gazebo which directly access the ODE core), so that exchanging the physics simulation will directly affect these parts of the simulation, too. Exchange or addition of methods for sensor simulation is usually easier, at least as long as these methods only require information already provided by the existing motion simulation. Several extensions for existing simulations have been described which add new sensors. Only OpenHRP though, provides a well defined interface for the exchange of simulation methods, in this case for the camera simulation.

2.3.4 Validation of Simulations

Following [126] validation of a simulation model is a process in which it is determined whether a simulation model is sufficiently accurate for a given application. Validation must cover three aspects of the simulation: appropriateness of the applied simulation methods for the given problem, sufficient accuracy of the modeling parameters and correctness of the implementation of the simulation. In the recent years validation of simulations for autonomous robots has become a topic of interest. Some current approaches are discussed in the following section.

Table 2.2: Comparison of sensor simulation capabilities.

Simulation	Camera Simulation				Distance Sensor Sim.	
	3D Rendering	Disturbances	Special Geometry	Abstract	Ray-casting	Depth-buffer
Blender	yes					
Gazebo	yes		(Omnidirectional) [a]	yes	yes	no
MSRS	yes			(yes)[b]	yes (method unknown)	
OpenHRP	yes[c]				no	no
Simbad	yes			no	yes[d]	no
SimRobot	yes	Rolling-Shutter Motion-Blur	Spherical Projection	(yes)[e]	yes[f]	yes
SimSpark	no			yes	no	no
Stage 2.1.1	no			yes	yes[g]	no
Stage 3.2	yes			yes	yes[h]	yes
Übersim	yes			no	no	no
UCHILSIM	yes	Noise Blur	Radial Distortion	no	no	no
USARsim	yes		Omnidirectional	limited[i]	yes[j]	no
Webots	yes	Noise		no	yes[k]	yes

[a] Described for two simulations based on Gazebo [112, 27], but not ported back into the main simulation.
[b] Can be implemented based on ground-truth data.
[c] May be exchanged through CORBA interface.
[d] Only sensor belts.
[e] Abstract sensor for object positions.
[f] Only single rays.
[g] 2D simulation limited to plane.
[h] Limited 3D support due to model of environment.
[i] Only certain simulation specific objects.
[j] Only single ray, fixed cones and 2D-sweeps.
[k] Only single rays or predefined distributions.

Systematic validation has only been done for few simulations for autonomous mobile robots. In this section some prominent examples for validation are discussed. To the author's knowledge, no approaches for the validation of a simulation on multiple levels of abstraction have been reported in the literature.

In [114] testing methods for the physical simulation of vehicles are presented and applied to validate the simulation of a skid steered robot in USARsim. The methods aim at choosing correct values for only three modeling parameters: the robot's center of mass, the friction and the angular damping (a parameter which determines the damping of the simulated robot's angular velocity). In a preliminary step experiments considering the friction and collision implementation of the simulation's underlying game engine were performed. After this, the performance of the simulated robot was compared with the performance of a real robot in a set of tests based on standard tests for autonomous robots developed at the NIST [105, 80]. These tests consider the robot's ability to overcome several obstacles. During these tests it was recorded which obstacles could (or could not) be overcome by the simulated and the real robot in the same manner. Based on the results of these comparisons the simulation methods as well as the modeling parameters were tuned to improve the accuracy of the simulation.

Besides validation of the motion simulation for USARsim also several sensor models have been validated. Validation of simulated cameras and wireless communication is described in [41]. The camera model is validated by comparing the performance of set of feature extraction algorithms for real and simulated images. The wireless simulation is validated by comparing the signal strength measured in a real world scenario and a simulation of the same scenario. In [20] the simulation of a GPS sensor has been validated by comparing the paths measured and the number of satellites seen by the real and simulated sensor.

In [104], simulations of a humanoid robot in USARsim and Microsoft Robotics Studio are investigated. Two experiments concerning the quality of the simulation based on USARsim were made. In the first experiment, the motion of the center of mass of the real and the simulated robot while walking along a straight line where compared. In the second experiment the real and the simulated robot executing the same behavior (in this case performing a penalty kick) were compared. In an additional experiment it was investigated how many robots could be simulated maintaining a realistic simulation.

To validate the simulation based on MSRS, experiments concerning the performance and the quality of the simulation are described in [104]: The performance of the simulation was evaluated by measuring the number of frames per second of the simulation for different numbers of robots. The quality of the simulation was evaluated by comparing different motion programs (with different complexity) for the simulated and the real robot. These comparisons where used to improve the modeling data. It is not revealed though, how the comparison was performed.

In [138], a 3D simulation model of a quadruped robot is validated for numerical optimization of the robot's motion parameters. Optimization of the motion parameters is done by solving an optimal control problem using the 3D simulation. Validation is based on comparing the joint angles of the real and simulated robot during several gaits. Results of this comparison are used to optimize modeling parameters for the motors (maximum torque and velocity) in an iterative procedure: After each modification of the model, the optimal control problem is solved, the resulting motion executed on the real robot and compared to the simulated motion.

In [94], a three step approach for the optimization of the parameters of a dynamic motion simulation of a four legged robot (a Sony Aibo) in the SimRobot simulation is presented. The method is based on comparing motions of a real robot and a simulated one to optimize the simulation parameters for the robot's model. Optimization is based on an evolutionary strategy. During the first step, the joint angles of a real robot performing a test motion while being fixed to a platform (and thus without ground contact of the limbs) are logged. These data are used to optimize the motor velocities and controller parameters of the model. In the second step again the joint angles of the robot are measured, but this time the robot is performing walking and soccer motions on the ground. These data are used to estimate the maximum torques of the motors. In a final step, the robot's velocity as well as the joint angles are measured while the robot is performing a set of walking motions. These data are used to optimize the modeling parameters concerning the robot's ground contact as well as parameters specific to the underlying ODE physics engine. In addition motor torques and controller parameters are further improved.

Discussion

The methods for motion simulation validation presented in the this section differ strongly in what is validated and how the validity of the simulation is evaluated. Considering the robot's motion, two different levels of detail are investigated: All approaches consider the motion of the robot relative to the environment, but only the approaches described in [138] and [94] additionally considers the motion of the robot's joints. The approaches also differ in how the motions of the simulated robot are evaluated. The approach in [114] only checks motions of the real and the simulated robot are performed in a comparable manner (e. g. if obstacles of the same height can or can not be overcome by the simulated and real robot alike). In the other approaches the robot's motion is regarded quantitatively. Besides considering the motion performed by a robot, the approach in [104] compares the way an autonomous maneuver is performed by the real and simulated robot, thus extending the scope of the validation to the sensor simulation.

Besides providing information on the accuracy of the simulation, all validation experiments were also used to improve the modeling parameters. Depending on the approach the parameters were tuned by hand or optimization techniques were used.

3 Proposed Methodology for Robot Simulations with Different Levels of Abstraction

In this chapter a methodology for robot simulation with different levels of abstraction is derived. Based on an analysis of typical applications of simulation in the context of autonomous robots, different levels of abstraction for motion and sensor simulation are identified in Section 3.1. In Section 3.2 a methodology is presented for the systematic analysis and validation of simulation methods for the same purpose on different levels of abstraction. The chapter closes with several design considerations for simulations with an adaptable level of abstraction in Section 3.3.

3.1 Requirements for Simulations

Depending on which modules and functionalities of a robot's control software are to be tested in simulation, different aspects of a robot's sensing and motion capabilities and interaction with the environment need to be simulated. Likewise demands on the accuracy of the simulations differ strongly. In this section, typical use cases for simulations and their respective requirements are presented. On this basis, levels of abstraction for sensor and motion simulation are identified and classified.

3.1.1 Typical Use Cases for Simulations

Testing of Single Sensor Processing

During sensor processing, readings of one sensor are refined to provide useful information on the environment or the state of the robot. The (usually noisy) readings of the sensor are transformed into a *percept* of the sensor. A percept is some kind of information which is initially represented in the sensor's coordinate frame.

How a percept is represented varies greatly with the kind of sensor and sensor processing. For an image sensor, "percept" may range from a blob of pixels to information on the position of specific objects of interest in the image plane. A laser range finder's percept may be some feature (e. g. a corner or an edge) or - again - the position of an object. The different kinds of percepts can be generalized into two categories: features (like blobs in an image or corners and edges found by a laser range finder) and objects of interest.

Depending on the kind of sensor several types of errors may impact the sensor-readings. Errors may be caused by the sensor itself (e. g. noise and saturation) or by the subsequent signal processing circuitry (e. g. limited resolution and saturation of an A/D converter). If the sensor is mounted on a mobile system, the motion of the sensor may further impact the sensor's readings (e. g. motion blur of a camera or Doppler shift of a microphone).

Even though a relatively simple simulation of a sensor's data without noise or other disturbances may be sufficient for testing basic algorithms for sensor processing, the simulation of the sensor's errors is crucial when investigating advanced algorithms handling noise and errors.

Testing of Multi Sensor Processing

Percepts provided by the processing of a single sensor's data are represented in a coordinate frame associated with the sensor. Usually percepts are transformed to a common coordinate system associated with the robot, especially if multiple sensors are used. If sensors are articulated (e. g. a pan-tilt-camera), a basic simulation of the robot's motion apparatus is required, as motion of the sensor will influence the transformation.

Testing of Self Localization and/or Mapping

Testing the self localization and/or mapping features of a robot requires simulation of the robot's motion within the environment as well as simulation of the robot's external sensors which are used for the task. Motion simulation must mainly provide information on the robot's position within the environment and, in case of articulated sensors, simulation of the sensor's motion. If the robot's motion has a significant impact on the data provided by the sensors, it is necessary to simulate these effects, too.

Requirements on the sensor simulation vary strongly, depending on the level of the tests as well as the algorithm in use. For basic testing of localization algorithms based on a-priori known landmarks, sensor simulation can be abstract, e. g. providing information on the landmarks' relative position, thus basically skipping any sensor processing. If, on the other hand, landmarks or other features are generated at runtime, as is the case for mapping algorithms, the sensors must be simulated with a level of accuracy sufficient to produce the artifacts the algorithm relies on.

Testing of Behavior Control

The task of a robot's behavior control is to decide the robot's next actions depending on the robot's knowledge of the current state of robot and environment. In a deliberative architecture, these decisions and plans are normally based on a world model the robot creates beforehand based on the robot's sensor readings. Output of the behavior control module are usually decisions on the robot's action on a high abstraction level, e. g. a motion vector, a target position or a request for a special motion.

Basic testing of the behavior control components of a robot's control software only requires simulation of the robot's motion and sensing capabilities on an abstract level. The sensor processing and world modeling stages of the control software may be skipped by providing ground truth data from the simulation. Depending on the modeling techniques used in the other stages it may be necessary to transform this information to the right reference frame.

Likewise the robot's motion needs not to be simulated accurately. Depending on the robot's mode of locomotion and the environment, it suffices to use an abstract 2D or 3D representation of the robot's pose. If the robot needs to interact with the environment, it is necessary to simulate this interaction, but like the rest of the motion a simplified model will suffice.

Using such simplified simulation methods has several advantages when testing behavior control:

- Scalability: Due to the low CPU consumption it is possible to simulate whole teams of robots, if necessary even faster than realtime, thus allowing extended testing of the behavior.
- Isolated testing: By skipping the sensor processing and world modeling stages, erroneous behaviors cannot be triggered by faulty sensor information but instead must have their origin within the behavior control.

If the correct functioning of the behavior control under optimal circumstances is ensured, advanced tests incorporating simulated errors in the input data may be performed. For a behavior control relying on abstract data (e. g. from the world model of the robot), an abstract simulation of errors (not a simulation of errors in the sensor data, but only errors in the world model which may be simulated much easier) is sufficient for the tests. Advantages of the use of a simulation in this context are the possibility to easily adjust the severeness of errors (e. g. by changing the noise level) and the repeatability of the experiments with *exactly* the same errors.

Testing Team Coordination and Communication

When testing the coordination of a team of robots, it is necessary to simulate all robots of the team at the same time. Depending on the robots in the team and the tasks to be fulfilled by the robots, different simulation methods may be required for each of the robots. As the focus lies on the coordination of the team, requirements on the accuracy of the simulation of the single robot's motion and sensors is not too high. If the coordination of the robots depends on some kind of communication, it is of interest to simulate the communication channel (and possibly communication errors). In this case it may be of interest to provide some information of the robot's motion, if motion may impact the communication.

Testing of Motion Planning

Motion planning for an autonomous robot is normally based on abstract information on the environment. Thus for testing the motion planning, it will suffice to simulate external sensors on a very abstract level. When testing high level motion planning, the main focus will be on the planned trajectory, thus an abstract simulation of the motion capabilities of the robot will be sufficient, too.

When testing low level motion planning (e. g. biped robot motion generation and stabilization) however, requirements on the simulation strongly differ. In this case, it will be necessary to provide an accurate simulation of the robots motion apparatus as well as of contacts with the environment. Another requirement is the accurate simulation of sensors used for the motion generation.

For robots with legged locomotion and manipulators,another relevant testing scenario is testing for basic correctness of the motion generation (e. g. avoidance of self collision during motion or for fulfillment of basic stability criteria like static stability). These tests require simulation of the whole motion apparatus, but only on a kinematic level.

Testing and Optimizing Robot Motion Generation

The motion generation module of a robot can be tested on several levels.

For a legged robot (or any other robot with an articulated mechanism, e. g. an arm or gripper) basic testing of the motion generation can be done to prevent damage of the robot due to unsuitable motions. When testing for self collision of the robot, simulation of the kinematic structure and collision detection will be sufficient to avoid most collisions. Advanced tests may consider the forces acting in the robot's joint, e. g. by applying algorithms for inverse dynamics, thus preventing overheating or destruction of the motors and gears. Such tests require a dynamics simulation of the robot as well as a sufficiently precise model of the robot's dynamical properties. None of these tests requires the simulation of the robot's sensors.

Different requirements hold for the model driven optimization of a legged robot's motion generation. In this case the simulation must provide a highly accurate model of the robot's motion dynamics as well as of the sensors that are fed back to the motion generation. Additionally, an accurate simulation of collision and contact of the robot's parts interacting with the environment is required.

Discussion

The examples given in the previous section show that the requirements on accuracy and abstraction level for a robot simulation highly depend on the use case. Requirements on the accuracy of motion and sensor simulation as well as specific requirements are summarized in Table 3.1. Based on the requirements identified in this section a hierarchy of abstraction levels allowing easier classification of different simulation methods will be developed in the following section.

3.1.2 Levels of Detail and Abstraction for Robot Simulations

Depending on the requirements for a simulation, it is desirable to use different levels of abstraction when simulating a robot's sensing and motion capabilities and its interaction with the environment.

In the following sections the different abstraction levels for sensor- and motion simulation are discussed.

Sensor Simulation

In a robot control application, the readings of the robot's sensors are usually transformed by a series of operations to create a representation of the state of the robot and the environment surrounding the robot. In Figure 3.1 several abstraction levels of sensor readings within a typical robot control application are depicted. This information can be used in further steps, e. g. the robot's behavior control, to decide the robot's actions.

Depending on which parts of the control application are to be tested with the simulation, it is desirable to provide sensor readings or the information derived from the readings on different levels of abstraction.

Table 3.1: Requirements for simulations.

use case: testing of ...	accuracy requirements for		other requirements
	sensor simulation	motion simulation	
... single sensor processing	medium (testing basic functionality) high (testing advanced algorithms)	low to medium	
... multi sensor processing	medium	low to medium	
... self localization and mapping	low (known landmark) medium to high (features created at runtime)	low to medium	
... behavior control	low	low	
... team coordination	low to medium	low	simulation of multiple robots robot communication
... motion planning		medium	
... basic motion generation	low	medium	collision detection
motion optimization	high (internal sensors) low (extenal sensors)	high	accurate simulation of collisions and contact

Readings of a robot's sensor can be simulated on several levels of abstraction:

- Sensor readings with sensor specific errors: The readings of the sensor are simulated using the representation of the real sensor. Additionally the sensor readings are distorted using a model of the real sensor's errors. Thus the control-software connected to the simulation must process the readings like readings from the sensor.
- Sensor readings with generic errors: perfect sensor readings are distorted using a generic model, e. g. for noise, saturation, or output characteristic. The control software must be able to handle these errors, but no handling of sensor specific errors is required.
- Perfect Sensor-Readings: The data read by the sensor are simulated, but not distorted. Thus the stages of the control software dedicated to pre-processing of the sensor data can be skipped.
- Percept-level: Instead of simulating the readings of the sensor, the information extracted from the readings is provided by the simulation. Depending on the kind of sensor, this can be very different kinds of percepts, e. g. positions of objects in a camera image or detected obstacles in a laser range finder's readings. The percepts can be represented in several different reference frames:
  - sensor centric: Percepts are represented in the sensor's reference frame. Percepts from different sensors still need to be transformed into a common reference frame, thus allowing the testing of this transformation. If percepts are simulated sensor-centric, it is also possible to investigate effects caused by the motion of the sensor (e. g. changing field of view).
  - robot centric: Percepts are represented in a common reference frame of the robot.
  - world centric: Percepts are represented in an external fixed reference frame. If this representation is to be of any use for the application, it also requires that the robot's pose within the same reference frame is known. Sensor simulation on this abstraction level is similar to ground truth data.

Examples for these levels of abstraction are given in Table 3.2.

Motion Simulation

Simulation for the motion of a mobile robot has to handle two different aspects of motion which may be considered with different accuracy within one simulation. The first kind is the motion of the robot's base within the environment. These motions will be called *outer motions* in the following discussion. The second kind is the motion of articulated parts of the robot relative to the base. Motions of this kind will be called *inner motions*.

Simulation algorithms for the outer motions of a robot classified by two categories:

- Dimension: A 2D simulation reduces the robot's motion to a plane, the robot's pose is described by three variables (two for the position and one for the orientation). 3D simulations allow the motion of the robot through the surrounding space. The robot's pose is described by 6 variables (three for the position and three for the orientation).

Figure 3.1: Abstraction levels of sensor readings- and information. Top: A soccer playing humanoid robot looking at a goal and the ball. Middle (from left to right): Image taken by the robot's camera, image after color segmentation, camera image (for better visibility in greyscale) augmented with sensor centric percepts of ball, goal and field lines . Bottom left: Percepts in robot centric coordinate system. Bottom right: world centric percepts (with additional information on the robot's estimated pose – which is slightly false in this image).

Table 3.2: Examples for abstraction levels in sensor simulation.

Abstraction level	Camera	Laser Scanner	GPS	Inertial sensors	Compass	Wheel- or Joint-encoders
1 Sensor with sensor specific errors	Simulation of effects of motion like blur or rolling shutter (e. g. [113])	Simulation of the sensor's motion; calculation of intersection at different times per ray; consideration of material dependent reflections	Consideration of motion and visibility of satellites (e. g. [21])	Consideration of missalignment of sensor's axes and crosstalk	Consideration of magnetic effects of objects surrounding the sensor	Nonlinearities of sensor, e. g. wear on potentiometers
2 Sensor with generic errors	Post processing image from renderer, e. g. applying gaussian blur	Applying noise to the length of the simulated length of the rays	Applying generic models for noise, sensor saturation and conversion artifacts			Discretization of incremental encoders
3 Sensor without errors	Image from 3D renderer	Ray intersections with object or depth buffer from 3D renderer	Position provided by motion simulation	Acceleration and angular velocity provided by motion simulation	Heading provided by motion simulation	Position of Joint; Velocity of wheel
4 Percepts (sensor centric)	Position of objects in image plane	Direction and distance of objects	Respective value in sensor's coordinate frame			/
5 Percepts (robot centric)	Direction of objects	Direction and distance of object	Respective value in robot's coordinate frame			/
6 Percepts (world centric)	Position of objects		Simulated integrated navigation solution based on motion simulation.			

- Kind of the simulation: Kinematical simulations allow setting the velocity and/or acceleration of the base and will simulate the robot's motion accordingly. Dynamical simulation will consider forces and moments acting in the robot's joints as well as forces and moments acting externally on the robot.

The inner motion of a robot can be simulated on three levels of abstraction:

- No simulation of inner motion: Omitting simulation of a robot's inner motions reduces the motion simulation to a point model. If no inner motions are simulated this limits the robot's possibilities for interaction with the environment to pushing objects. As it is not possible to simulate the motion of sensors independently of the motion of the robot's base, it is not possible to consider the absolute position of a sensor, thus sensor readings only can be simulated in a robot centric coordinate frame.

- Kinematical simulation of inner motion: This kind of simulation enables examination of the robot's interaction with the environment (e. g. grabbing an object or kicking a ball). It also enables the consideration of articulated sensors which may move independently of the robot's base, thus allowing for the sensor centric simulation of sensor readings.

- Dynamical simulation of inner motion: This enables the consideration of whole body motions of the robot.

It should be noted, that simulation of inner motions may be 3D, even if the outer motions are simulated in 2D only. This approach may be used if the robot's outer motion is confined to a plane (e. g. for a ground vehicle) and a simplified model of the motion suffices while still motions of the robot's sensor are to be considered. Likewise it may be of interest to combine dynamical simulation of the robot's outer motion with kinematical simulation of the inner motion of the robot, e. g. if dynamic effects of articulated sensors can be neglected compared to the dynamics of the robot's motion within the environment.

Depending on the level of accuracy of the motion simulation a sufficient consideration of the robot's contact situation is required. Again this can be handled on several levels of detail.

- No collision detection: If the algorithm for motion simulation is based on some kind of assumption, e. g. confinement of the motion to a plane, no collision detection is required to support the motion simulation. In this case, though, there can be no further interaction of the simulated robot and environment.

- Collision detection for selected parts of the robot: In many cases, only some specific parts of a robot can be in contact with the environment. If no other contacts may occur (or are of no interest) in a use case, collision detection can be cut down to these parts. A typical use case for this simplification is a simulation regarding only the locomotion of a biped robot. In this case, collision detection only needs to consider the feet, leaving more CPU time for other tasks, e. g. elaborate dynamics simulation or expensive ground contact models.

- Collision detection for all parts of the robot: Full collision detection is required if the simulated robot's interaction with the environment is not limited to some parts (e. g. if the robot is capable of falling down, getting up or rolling over).

Independently of the consideration of collisions with the robot's environment, collisions of the robot with itself (from now on called *inner collisions*) may or may not need to be detected. If

no inner motion of a robot is considered in a simulated scenario, obviously detection of inner collisions is not required. On the other hand detection of inner collisions is crucial, if the robot's motion are the main scope of the simulation, e. g. when developing stable walking motions for a legged robot.

3.2 Validation

As discussed in Section 2.3.4, validation of a simulation has to cover the correctness of the implementation, the appropriateness of the simulation method for the selected application and the accuracy of the modeling parameters used. In this section a methodology for the structured selection of methods for the validation of simulation method and modeling parameters is presented. This method is based on an analysis of the requirements for the specific simulation and the available simulation methods. By this it provides a guideline in selecting validation experiments which will lead to a validation at the proper level of accuracy. Validation of the correctness of the implementation will only be covered briefly.

As shown in the previous section, requirements on abstraction and level of accuracy of a simulation strongly differ depending on the use case. If several methods for the same aspect exist in the same simulation, validation of these methods and the modeling parameters can be used to choose a method adequate for the respective task.

3.2.1 Validation of Implementation

Before even trying to validate a simulation method or modeling parameters, it is important to make sure that the implementations of the methods used in the simulation are correct. Due to the complexity of many simulation algorithms a formal verification often is not possible.

Instead, this methodology suggests the use of unit tests (e. g. [74]). Unit tests are well suited to the implementation of simulation methods, as each method can be tested separately. As discussed in Section 2.2 and 2.3, often different simulation methods exist for the same purpose (e. g. the CRBA and the ABA for the calculation of a robot's forward dynamics). If several such methods are implemented, unit tests can be easily reused. As unit tests can only test for a set of predefined test cases, it is suggested to additionally use pre and post conditions within the implementation of a simulation method. These can be used to test the implementation during runtime.

3.2.2 Validation of Simulation Methods and Modeling Parameters

The way a validation of a simulation is performed strongly depends on the level of abstraction and accuracy provided by the simulation method used during validation. In general, it will not be possibly to validate a simulation once for all purposes. Whether a simulation is valid for a given purpose, always depends on the specific requirements of that purpose. To decide whether the validation is sufficient, requires knowledge of the requirements of the specific purpose as well as the possibility to check these requirements with respect to the results of validation.

To structure the validation process and to allow a later comparison of simulation methods on different levels of abstraction and accuracy, an iterative four step method is proposed:

1. Identification of simulation effects.
   This step can be summarized with the question "What must be simulated?". To answer this question, it is broken further down. At first, the aspects that need need to be simulated and validated are collected. After this, effects that affect these aspects and are relevant are identified. This step leads to a list of phenomena and related effects which are of interest.

2. Evaluation of simulation models and methods.
   This step can be summarized with the question "How is it simulated?". In this step, the available simulation approaches (consisting of models and methods) for the aspects and effects identified in the first step are evaluated. Evaluation may consider different parts of the simulation, e. g modeling parameters (like kinematic or dynamic parameters of the robot), modeling approaches (like the approach used to solve the robot's equations of motion) or computational methods used for solving the equations of motion.

   For the purpose of evaluation, it is checked for each available approach which of the aspects and identified effects are covered. Coverage is evaluated by three categories for each effect:

   - not applicable: The effect cannot be simulated.
   - qualitatively comparable: In principle the effect can be simulated, so that it behaves in a plausible way (but not necessarily quantitatively the same as in the real robot).
   - quantitatively comparable: In principle the effect can be simulated in a way that can be compared to reality on a quantitative level.

   The distinction between qualitative and quantitative comparability is made to reflect the different requirements on a simulation's accuracy and abstraction level which were identified in the previous section.

   For each effect which can be simulated by a given approach, all relevant parameters of the simulation model that have influence on this effect are summarized to guide validation of model parameters in subsequent steps. This step leads to a crosstabulation of aspects/effects and simulation methods. For each pair it is noted, on what level of accuracy it can be simulated and which modeling parameters are relevant.

3. Selection of validation methods.
   In this step validation methods are selected which are suitable to validate the simulation of an effect or aspect, respectively, on the abstraction level provided by the simulation method. Which validation method to select highly depends on the respective simulation method and whether it is to be validated qualitatively or quantitatively.

   When designing validation experiments, care must be taken to ensure that the effects of interest are of importance for the robot's behavior in the respective experiment and can be properly observed. Otherwise the impact of a specific effect may be masked by other effects. An example is the motion of a legged robot, which can be influenced by different effects like slipping of the feet and elasticity in the joints. If a simulation is capable of simulating either effect, experiments should be designed which allow separate observation of the effects. Even though the overall influence of elasticity and slipping on the whole robot's motion can be determined by only measuring distances covered by the robot, an experiment which measures data on each of the robot's joints will lead to additional information on the elasticities which can be evaluated independently from the slipping.

4. Validation Experiments.
   In this step the previously selected experiments are performed. Two objectives are followed by these experiments:
   - It is checked, whether the given method can be used to simulate the effect/aspect under consideration. Depending on the level of abstraction of the method, this can be a simple judgement if the method yields qualitatively comparable behavior or a quantitative analysis of the behavior. The results of this check are collected to create a crosstabulation of methods and aspects/effects stating for each pair if, and on what level of accuracy, it can be simulated.
   - Additionally, the validation experiments can be used to optimize parameters of the simulation to improve the behavior of the simulation. This can be done using optimization techniques, e. g. a non-linear least squares approach which minimizes a weighted sum of the squares of the differences between simulated and experimentally measured robot behavior.

   Depending on how many effects affect a specific aspect of the simulation, experiments may be made directly investigating the aspect or may start on the level of effects. Especially if some effects only depend on a subset of the modeling parameters identified, it is helpful to start with these effects in order to estimate some of the parameters.

   During validation experiments it may become obvious, that the phenomenon under consideration is affected by further effects. This will lead to a refinement of the data collected in the previous steps.

3.2.3 Selection of an Adequate Simulation Method

The crosstabulation created in the last step of the validation methodology can be used to evaluate simulation methods for their fitness for a given purpose. To do this, it has to be evaluated, if a given (validated) level of abstraction is sufficient for the purpose. This allows an objective comparison of the simulation methods and may lead to several conclusions:

- One or more simulation methods are accurate enough for the given purpose.
- If a method lacks accuracy in some points, this enables a discussion, whether the method can be used anyway (and which problems may occur from using it).
- Additional effects may be found that haven't been validated yet. This will lead to another set of experiments refining the validation table.

3.3 Design Considerations

3.3.1 General Structure

Simulation of mobile autonomous robots requires several simulation methods (e. g. for sensors or the robot's motion) to act in concert. For a simulation with adaptable levels of abstraction and accuracy, several methods for the same general purpose may exist and it may be of interest to add further methods to an existing simulation. These considerations lead to three mayor requirements on the general design of the simulation:

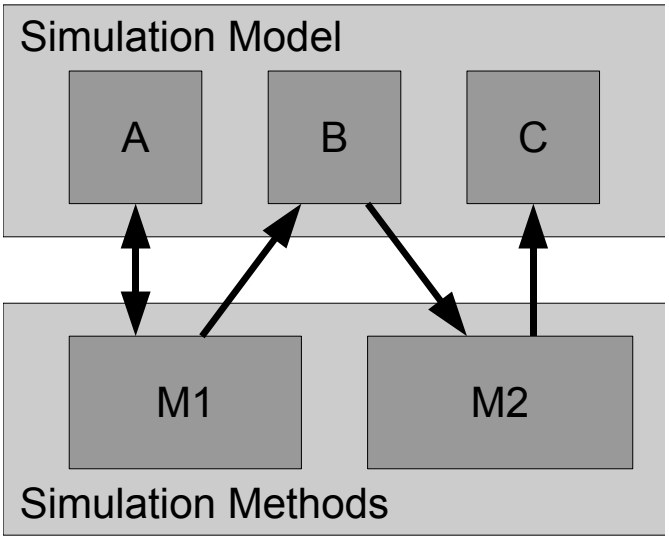


Figure 3.2: General structure of a simulation. In this example two simulation methods are applied. Each of the methods reads and writes some of the elements of the model. One of these elements is accessed by both methods, thus enabling data exchange between the methods without either method being aware of the other.

1. Strict separation of the representation of the robots (modeling parameters like kinematical structure or dynamical properties as well as the representation of the current state like velocities, external forces etc.) and the implementations of simulation methods:
   As it is not known beforehand which simulation methods are to be used, it is not feasible to store any data which might be of interest to other methods within an internal representation of any specific implementation. Otherwise each implementation of any other method will depend on the interface of each existing implementation.

2. No fixed interfaces for methods with the same purpose:
   If simulation methods for the same purpose are provided on different levels of abstraction, these methods will require access to different properties of the modeling data. Thus it is not feasible to provide one single interface for all methods with the purpose. This requirement does not inhibit the use of interfaces. If several methods are to be implemented that access the same data, it will still be most helpful to provide a common interface for these methods. But it is not desirable to impose this interface onto all implementations.

3. No fixed data structures for modeling the robot:
   Newly developed simulation methods may require more or different modeling information and may likewise simulate new properties which were not thought off when creating the initial models of the simulation. Thus it must be possible to add further information later on without breaking existing implementations of simulation methods.

These considerations lead to a simple general structure of the simulation. The simulation consists of models representing the robots and the environment and several simulation methods connected to these models. As it is not known beforehand, which simulation methods will be incorporated into one simulation, any data exchange between these methods has to be done via the models, see Figure 3.2.

3.3.2 Execution of Simulations

The consideration of the previous subsection leads to three distinct phases for setting up and running a simulation.

1. Modeling: During the *modeling phase* the models representing the robots present in the simulation as well as for the environment are created. In this phase, the relation of objects and all *constant* parameters of the models are defined.

2. Algorithm setup: In the *setup phase* instances of the simulation methods used for the current simulation are created and linked to the models. Additional *variable* properties needed as input or produced as output by any of the algorithms are added to the data-model created in the preceding modeling phase. Besides algorithms calculating the progression of time, further algorithms may be used to communicate with the control software of the simulated robots.

3. Simulation execution: During the *execution phase*, the simulation algorithms are executed. Depending on the requirements on the simulation, the execution rates of single algorithms may vary (e. g. higher rates for motion simulation than for sensor simulation).

3.3.3 Discussion

The described structure of the simulation has several advantages for the creation of simulation with an adaptable level of abstraction.

Most important, it is possible to exchange simulation methods transparently. As long as a method provides all information needed by other methods applied in the same simulation, it is not of interest, on which level of abstraction or accuracy the specific method is working.

Further on it is known for each method applied, which data elements are read and/or written by each instance of the method. This allows for an easy parallelization of the simulation on multiple CPUs.

The methodology discussed here has been used as a base for the development of the Multi-Robot-Simulation-Framework (`MuRoSimF`). In the following three chapters the main aspects of this framework, namely the modeling of robots for simulation, simulation methods on different levels of abstraction and the execution of simulations are discussed. After this, in Chapter 7, several simulation created with `MuRoSimF` are presented and discussed.

4 Adaptable Modeling of Robots

In this chapter the concepts used to represent simulation models of robots within `MuRoSimF` are presented. The objective of these concepts is to provide a clean separation of the models describing the simulated robots and the simulation methods used. The modeling methodology has not been designed with any specific simulation method in mind, but instead allows the use of models with any simulation method for general tree structured robots as well as the use of specialized methods making more assumptions on the robot's structure.

Robot models are composed of a small set of basic elements describing the robot's structure. Additional elements like sensors can be attached to this structure in any place. Mechanisms are provided to ensure that properties calculated by the motion simulation are transparently made available to sensor-elements later on attached to the structure.

4.1 Modeling Hierarchy

Modeling in `MuRoSimF` is based on a hierarchy of data structures (cf. Figure 4.1). Robot systems and the simulated environment are described as so called *compound objects*. Each compound object is a collection of single *objects*. The object's within one compound object may be unstructured or of a model dependent structure (e. g. tree-structured). Each object is a collections of *properties* describing the object.

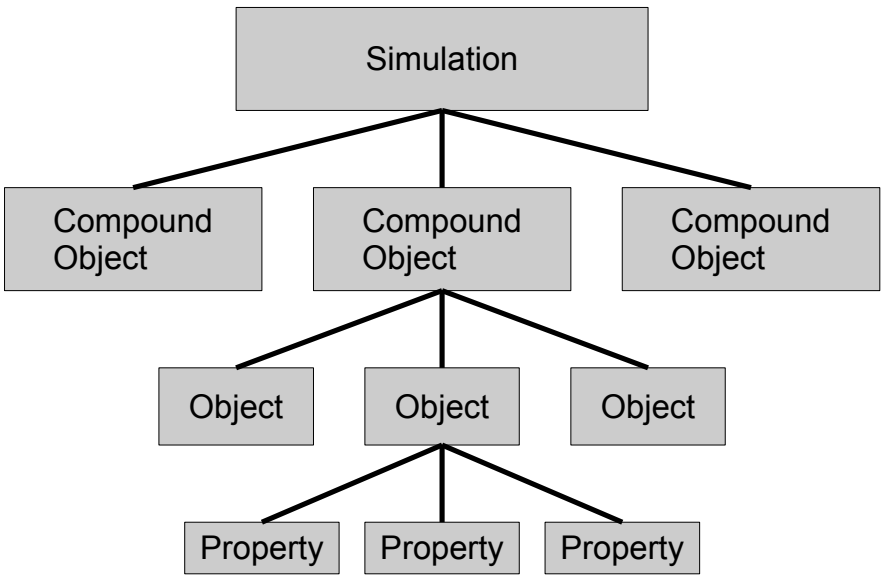


Figure 4.1: Hierarachy of the data objects stored within a simulation.

4.1.1 Objects

Each object within a simulation is described by a set of properties. An object provides methods to add properties to the object and to check, whether a specific properties is present for this

Table 4.1: Examples of properties added to objects during model setup.

Property	Symbol	Type	used by
Mass	m	$\mathbb{R}$	Dynamics algorithms
Center of mass	com	$\mathbb{R}^3$	
Inertia tensor	I	$\mathbb{R}^{3\times3}$	
Shape			Collision detection / Visualisation
Surface properties			Collision handling
Color / Texture			Visualisation

Table 4.2: Examples of properties added by algorithms to the objects

Property	Symbol	Type
Position	$\mathbf{r}$	$\mathbb{R}^3$
Orientation	R	$\mathbb{R}^{3\times3}$
Velocity	$\mathbf{v}$, ω	$\mathbb{R}^3$,$\mathbb{R}^3$
Acceleration	$\dot{\mathbf{v}}$, $\dot{\omega}$	$\mathbb{R}^3$,$\mathbb{R}^3$
Joint position	q	$\mathbb{R}$
Joint rate	$\dot{q}$	$\mathbb{R}$
Joint acceleration	$\ddot{q}$	$\mathbb{R}$
External force	$\mathbf{f}_{\mathbf{ext}}$	$\mathbb{R}^3$
External torque	$\mathbf{n}_{\mathbf{ext}}$	$\mathbb{R}^3$

object. The later method is used during algorithm setup of the simulation to decide, whether a specific simulation method may be used for an object.

4.1.2 Properties

During setup of a model constant properties are added to the objects describing a model's structure and other invariant properties. These modeling parameters are independent of the algorithms used later by the simulation (cf. Table 4.1). Later on when simulation methods are connected to the model, variable properties needed as input or produced as output by the respective algorithm are added to the objects (cf. Table 4.2).

Explicit Equality

Under special circumstances some of the variable properties of a set of objects may be constantly equal (e. g. the orientation of two coordinate frames connected by a rigid translation). Normally this equality is expressed by the simulation algorithm by assigning values in every timestep of the simulation. To avoid this repetitive assigning of the same value, it is possible to declare a property to be equal for a set of objects. After this, all objects of this set will share the same memory for storing this special property, thus enforcing equality. Using this feature, objects representing sensors can be attached to other objects of a scene sharing the properties needed for sensor simulation. Another use of this feature is made of in the implementations of several simulation methods (e. g. Section 5.1.1 and Table 5.2).

4.1.3 Compound Objects

Complex systems consist of a set of independent objects. Such a set of objects belonging together will be referred to as a *compound object*. A basic compound object has no inner structure of its members. In specialized types of compound objects though, a structure can be imposed on the objects. In a basic compound, it is possible to access each named object by name. Further on, it is possible to access a list of all members of compound object.

Subsets and Distinguished Elements

Besides its structure, a specialized compound object can be further described by defining its *distinguished objects* and *subsets*. Distinguished objects are objects which have special properties or a special position within the structure, e. g. the feet of a biped robot. Subsets are sets of objects which share some kind of property or structural feature, e. g. the subset of a robot's joints. Further examples of subsets and distinguished object are given in sec. 4.3.

4.2 Views

During execution of a simulation, several simulation algorithms will access data from the models used. Each of these algorithms only needs access to some aspects of the model. Further on, only some of this data needs to be written. This situation is represented in `MuRoSimF` by so called *views*, cf. Figure 4.2.

Two different kinds of views are distinguished:

Views on single objects allow access to a well defined subset of the properties of the object connected to the view. When creating a new view, it is connected to the object. This connection cannot be broken later on. When connecting to the object, the view checks, if the object has all properties required by the view. Additional properties may be added by the view to the object. The view will be valid, if all required properties are present in the object. If only reading or also writing access is possible, can be defined for each property individually. Views may implement functionality beyond simple reading and writing of properties. This may be useful if complex operations have to be performed.

Views on compound objects allow access to selected objects and subsets of a compound object. There exist different mechanisms for the selection of the objects. The most basic way is to select objects by their respective names. It is also possible to select a subset of the compound consisting of all objects for which a given single object view if valid. When creating a view on a compound object, it will be connected permanently to the compound object. The view will be valid, if all required objects are present in the compound object.

The use of views is not mandatory, but it is strongly recommended. Algorithms also can directly access an object and query for the required properties. Views simplify the design of simulation algorithms, as they provide a well defined interface to the model data. Using a view it can be determined easily, if an object has all properties required by a given algorithm, thus separating the implementation of the algorithms from the problem of checking a specific object. If several different algorithms exist for solving the same simulation task, the view may be reused, thus supporting transparent exchange of algorithms.

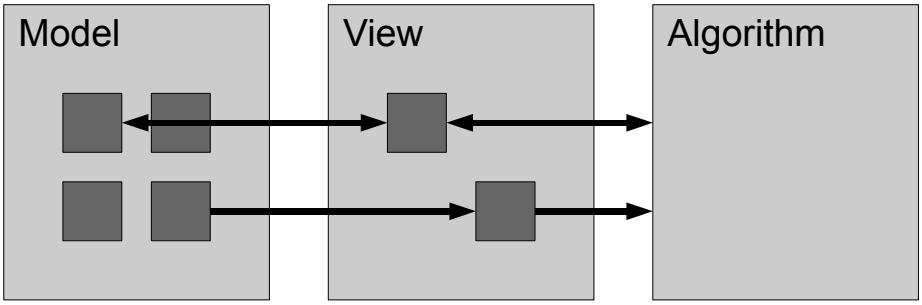


Figure 4.2: Interaction of model, view and algorithm.

Table 4.3: Elements for modeling the kinematical structure

Element	described by	Symbol	Type
Base			
Fixed translation	translation vector	${}^{\mathrm{pred}(i)}\mathbf{r}_i$	$\mathbb{R}^3$
Fixed rotation	rotation matrix	${}^{\mathrm{pred}(i)}R_i$	$\mathbb{R}^{3\times3}$
Variable translation	translation direction	$\mathbf{e}_{\mathbf{t}i}$	$\mathbb{R}^3$
Variable rotation	direction of axis	$\mathbf{e}_{\mathbf{r}i}$	$\mathbb{R}^3$
Fork			
Endpoint			

4.3 Modeling of Robot Systems

In this section it is described, how different robot systems can be modeled using `MuRoSimF`.

4.3.1 Multi Body Systems

The foundation for modeling robots in `MuRoSimF` are tree structured multi body systems (MBS). An MBS-model is a specialized compound object providing a tree-shaped organization of the objects and further MBS specific information. In [76], a very general object oriented approach for the decomposition of the structure of a MBS as been presented. This approach has been simplified and specialized for the needs of the implementation of `MuRoSimF`.

Kinematical Model

Modeling of the kinematical structure of a robot is based on seven basic elements: base, fixed translation and rotation, variable translation and rotation, fork and endpoint, cf. Table 4.3.1. The base will be the root node of the kinematical tree, it has no predecessor and one successor, rotations and translations each have one predecessor and one successor, forks have one predecessor and two successors and points have one predecessor but no successor.

Variable rotations and translations are the foundation for modeling a robot's joints. If the joint position q is added to the object representing a joint, it is possible to use algorithms calculating the robot's *direct kinematics* (cf. 2.2.1 and 5.1.1), thus enabling the calculation of position $\mathbf{r}$ and orientation R of each element of the MBS-tree.

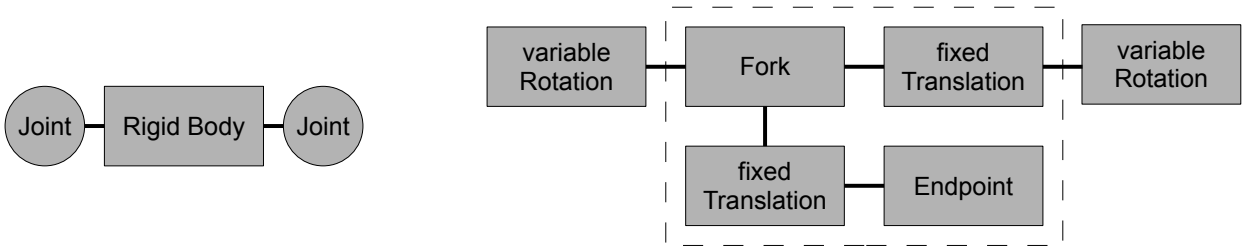


Figure 4.3: Left: Part of chain shaped kinematic structure consisting of two joints connected by a rigid right. Right: Modeling of the same structure in `MuRoSimF`. The rigid body is decomposed into four parts with the upper translation describing the kinematic structure of the body and the lower translation and endpoint describing the bodies dynamical properties.

Dynamical Model

The kinematical model can be extended to the dynamical model by assigning further properties to the elements of the tree. To describe a rigid body model, these properties are the mass, the center of mass and the inertia tensor of each body.

To facilitate the task of implementing simulation algorithms three restrictions are made in `MuRoSimF`.

1. All properties used by the dynamics calculations may only be stored in the endpoints.
2. All properties are represented in the coordinate system of the respective object.
3. The center of mass is always in the origin of an object's coordinate system.

These restrictions lead to a model where a single rigid body is described by a set of modeling elements, often including bifurcations in the structure, see Figure 4.3. A major benefit of this approach is, that the implementation of MBS-algorithms is facilitated: The algorithms can be broken down into sub-algorithms for each of the modeling elements. Each sub-algorithm can be implemented and tested individually. As it is known beforehand, that certain properties do not occur in some modeling elements, implementations for these elements will be simpler than general implementations.

To simulate dynamical properties of the motor driving the joints of an MBS model, additional properties describing gears, rotor inertia and friction can be added to a joint. Note that for more complex modeling further properties can be added easily to a model.

An overview of the additional properties for the dynamical model is given in Table 4.4.

Distinguished Elements and Subsets

A MBS-model provides several distinguished objects and subsets which can be used for the formulation of algorithms:

- Base: the root element of the MBS-tree.
- Joints: all variable rotations and translations of the MBS-tree.
- Bodies: all endpoints of the tree which can experience external forces.

Table 4.4: Additional properties for modeling the dynamics of an MBS.

Added to Element	Property	Symbol	Type
Endpoint	Mass	m	$\mathbb{R}$
	Inertia Tensor	I	$\mathbb{R}^{3x3}$
Joint	Viscous friction	μ_f	$\mathbb{R}$
	Motor inertial	J	$\mathbb{R}$
	Gear ratio	ρ	$\mathbb{R}$

4.3.2 Robot Systems

Robot systems are described by extending the tree-structure of the multi-body-system with additional features like actuators, sensors or shapes. Following the design paradigms presented in the last section, shapes may only be added to endpoints of the kinematic structure, as they may experience external forces by collisions. In addition to the subsets provided by any MBS-model, a robot has two more subsets: *actuators* and *sensors*.

Actuators

Actuators are special objects which can be attached to a robot's joints. When attaching an actuator-object to a joint, the actuator-object will *share* the properties describing the motion of the joint (position q, velocity $\dot{q}$, acceleration $\ddot{q}$ and force/torque τ) using the *explicit equality* mechanism (see Section 4.1.2). The actuator-object may store additional properties depending on the kind of actuator used, e. g. properties for the electrical specification of the motor used in a servo-motor. Using actuator-objects (instead of storing the additional actuator related properties in the associated joint object), allows a differentiation of the use of the joint and the actuator-objects. Joint-objects are part of the MBS-model and are used for MBS-related simulation methods (e. g. dynamics simulation of a robot's motion). Actuator-objects are not part of the MBS-model (and thus not known to the MBS-related parts of the simulation). They are used for simulation of the actuators driving the joints of the underlying MBS-model. The connection of both parts of the simulation are the common properties used by the joint and actuator. This approach allows a decoupling of the simulation of the robot's motion (by an MBS method) and the simulation of the actuator, e. g. a servo-motor's control strategy.

Sensors

Sensors are objects which can be attached to any object of a robot's structure. When attaching a sensor to another object, the sensor will share all of the object's properties relevant for simulating the sensor. For each kind of sensor, specialized subclasses exist, defining which properties are to be shared and which additional properties are needed (e. g. for modeling sensor errors). An overview of properties relevant for sensor simulation is given in Section 5.4.

Using sensor-objects instead of other objects (with additional sensor related properties) allows adding an arbitrary number of sensors to any element of the robot. This is especially helpful, if several sensors of the same type have to be added to the same element (e. g. three one-axis acceleration sensors for a simulated 3D measurement).

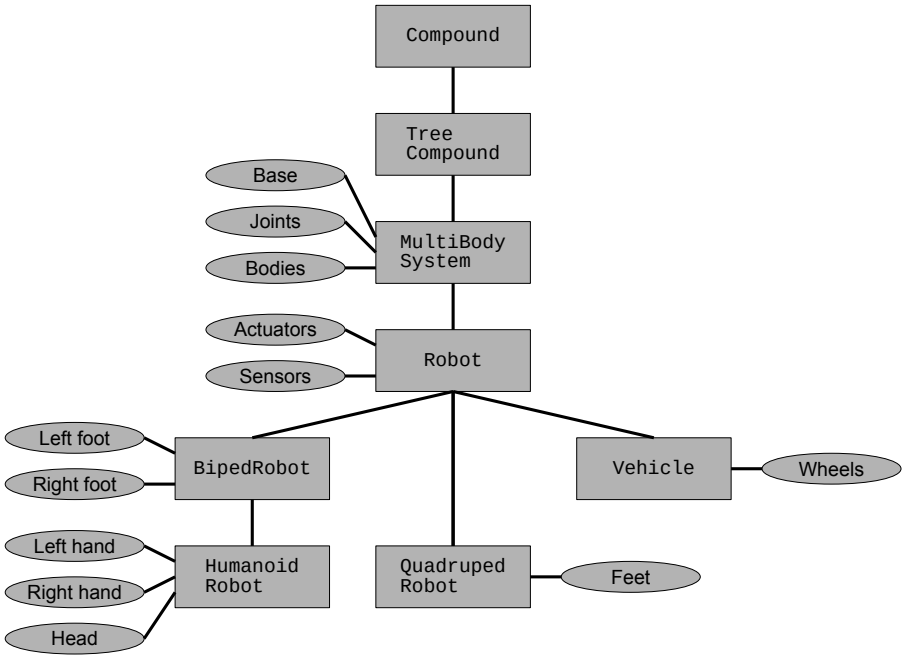


Figure 4.4: Hierarchy of models. Subsets and distinguished objects are depicted as ellipses linked to the level of the hierarchy where they are introduced.

Specialization

Depending on the structure of the robot modeled, further subsets and distinguished objects may be defined to support the implementation of robot specific algorithms. An overview of typical structures and their relation is given in Figure 4.4. This hierarchy can be implemented in two ways: One possibility is a hierarchy of more and more specialized compound objects, the other possibility is to expose the distinguished objects through a hierarchy of views, where a special view only is valid if a compound has all required subsets and distinguished objects.

4.3.3 Defining Robot Models

Two approaches are provided to define robot models.

Definition in C++

Robot models are defined using a class called `RobotModelBuilder`. This class provides methods for successively adding elements of the kinematic structure depth first, starting with the base. For each element specific properties may be set during modeling using respective methods after adding the structural element.

The kinematic structure of a simplified biped robot with 6 DOF can be defined by code fragment given in Listing 4.1. The resulting robot is shown in Figure 4.5. Note that for the sake of simplicity, only the mass properties for the robot's base and feet have been set. Other properties like the inertia tensor, the shape or visible appearance can be set in the same manner.

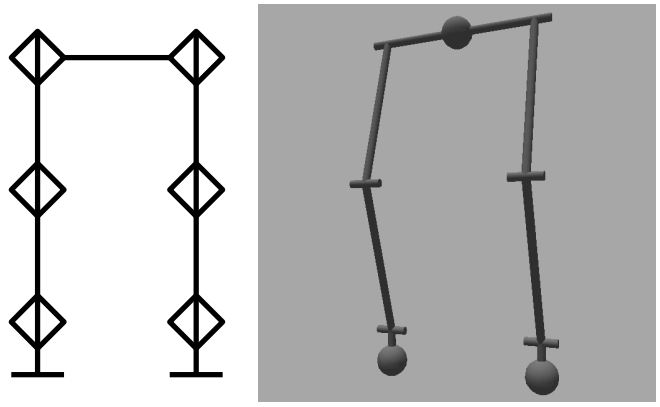


Figure 4.5: Kinematic structure an example robot defined in listings 4.1 and 4.3. Left: schematic model, Right: model in simulation.

Using the definition in C++ allows an easy integration of a model into a simulation. As it is known, which kind of robot is created, distinguished elements and subsets of the robot can be set easily in a specialized subclass of `Robot`. For convenience a `RobotModelBuilder` can be added easily to an objects constructor, thus allowing to setup a model within its own class (see listing 4.2) Any changes to the model require a new compilation of the simulation.

Definition at Runtime

To allow for an easier redefinition of robot models, a description language has been developed which allows a depth-first definition of the robot as well as setting additional properties of the robot's elements. The basic structure of a definition is similar to a definition in a C++ program. Robot definitions can be parsed at runtime from a file, allowing the easy change of robot models. As the parser has no knowledge of the special type of robot described, it will always generate an object of the type `Robot`. If algorithms for specialized robots are to be used on such models, information on distinguished objects or subsets must be obtained by specialized views extracting such elements by predefined names.

Using the description language the example robot from Figure 4.5 can be defined using the description in listing 4.3.

4.4 Discussion

Most of the existing simulators for autonomous robots presented in Section 2.3 have been designed on the base of an existing simulation package, usually providing a specific method for the robot's motion simulation. From this starting point it is an obvious choice to chose a model representation fitting (and often limited) to the simulation methods provided by this package. By this choice though, the simulation will be limited to those properties provided by the specific simulation method. If further properties need to be simulated, this will require working around the specific limits.

In this section a different methodology for modeling robots for simulations has been presented. Unlike the modeling paradigms used by most other simulations, this methodology is not limited to a given set of properties which are defined by a set of simulation methods known

Listing 4.1: Modeling a simplified robot using C++.

```
// an instance of RobotModel for the new model
RobotModel model("BipedRobot");
// a builder to create the model
RobotModelBuilder b(&model);

// modeling starts with the robot's base ...
b.addFreeBase("base");

// ... a fork is added for a body connected to the base ...
b.fork();
// ... the body is defined ...
b.addPoint("centralBody");
b.setMass(0.3);

// ... after this a fork for the legs is added ...
b.addFork();

// ... the left leg is modeled ...
b.addTranslation(0,0.1,0,"leftHip");
b.addRotationalJoint(0,1,0,"leftHipJoint");
b.addTranslation(0,0,-0.2,"leftUpperLeg");
b.addRotationalJoint(0,1,0,"leftKneeJoint");
b.addTranlation(0,0,-0.2,"leftLowerLeg");
b.addRotationalJoint(0,1,0,"leftAnkleJoint");
b.addTransation(0,0,-0.04,"leftAnkle");
b.addPoint("leftFoot");
b.setMass(0.2);

// ... and then the right leg is modeled.
b.addTranslation(0,-0.1,0,"rightHip");
b.addRotationalJoint(0,1,0,"rightHipJoint");
b.addTranslation(0,0,-0.2,"rightUpperLeg");
b.addRotationalJoint(0,1,0,"rightKneeJoint");
b.addTranlation(0,0,-0.2,"rightLowerLeg");
b.addRotationalJoint(0,1,0,"rightAnkleJoint");
b.addTranlation(0,0,-0.04,"rightAnkle");
b.addPoint("rightFoot");
b.setMass(0.2);
```

Listing 4.2: A specialized model building itself within the constructor.

```
// Declaration of the class.

class MyHumanoid:public HumanoidRobot{
public:
  //Constructor.
  MyHumanoidRobot();

  //....
}

// Definition of the class.
MyHumanoidRobot::MyHumanoidRobot(){
  // create a builder on the object itself
  RobotModelBuilder builder(*this);

  // start with the base ...
  builder.addFreeBase("base");

  // ... and continue defining the robot.

  // ...
}
```

Listing 4.3: Modeling a simplified robot using the description language.

```
% modeling starts with the robot's base ...
freebase base

% ... a fork is added for a body connected to the base ...
fork *
% ... the body is defined ...
point centralBody
mass 0.3

% ... after this a fork for the legs is added ...
fork *

% ... the left leg is modeled ...
trans leftHip 0 0.1 0
rotjoint leftHipJoint 0 1 0
trans leftUpperLeg 0 0 -0.2
rotjoint leftKneeJoint 0 1 0
trans leftLowerLeg 0 0 -0.2
rotjoint leftankleJoint 0 1 0
trans leftAnkle 0 0 -0.04
point leftFoot
mass 0.2

% ... and then the right leg is modeled.
trans rightHip 0 -0.1 0
rotjoint rightHipJoint 0 1 0
trans rightUpperLeg 0 0 -0.2
rotjoint rightKneeJoint 0 1 0
trans rightLowerLeg 0 0 -0.2
rotjoint rightAnkleJoint 0 1 0
trans rightAnkle 0 0 -0.04
point rightFoot
mass 0.2
```

a-priori. Instead it has been designed to provide a high level of flexibility when modeling robots for a simulation, thus providing the base for the flexible and transparent exchange of simulation methods. This flexibility can be found on two distinct levels:

On the *structural level* a set of currently seven modeling primitives is provided to create a robot's kinematical structure. If new simulation methods require further primitives (e. g. elastic- or ball-and-socket-joints) or allow different structures (e. g. closed loop kinematics) these can be added without influencing existing models. Obviously models using new primitives or structures also require a simulation method capable of handling these.

On the *object level* it is possible to add any property which is of interest to a simulation method to each object of the robot's model. By this it is possible to extend an existing model with new properties which have not been of interest when creating the model in first place. If new properties are introduced to the simulation, these can be shared among several methods, as all methods can access the common storage of the model.

This flexibility of robot modeling is not present in typical modeling paradigms like the well known Denavit-Hartenberg-convention [45] which was designed with a specific application in mind (like manipulator kinematics). The modeling approach is also more flexible than the models in simulations that were created around some simulation package.

The concept of views allows easy definition of interfaces for specific simulation methods, so that the implementation of the method does not need to directly interface with the model's representation (thus enabling easy exchange of the method's implementation). At the same time, it is possible to define any interface to the modeling information, so that newly developed methods are not limited to interfaces present, as it is the case in some of the existing simulations (e. g. Webots or OpenHRP, see Section 2.3.3 for more details).

5 Computational Methods for Simulation on Different Levels of Abstraction

In this chapter, simulation methods for the motion and sensors of mobile autonomous robots on different levels of abstraction are presented. Within MuRoSimF it is possible to combine implementations of different methods within one simulation allowing the flexible adaption of the simulation to different requirements.

5.1 Robot Motion Simulation

The purpose of the methods presented in this section is the simulation of a robot's motion within its environment. In the following subsection, basic algorithms for robot motion algorithms and their implementation within MuRoSimF will be presented. After this several advanced algorithms on different levels of abstraction are discussed. This discussion covers specialized algorithms restricted to specific robot structures as well as general algorithms.

5.1.1 Basic Algorihms

Basic recursive MBS algorithms for motion simulation like direct kinematics, RNEA or the ABA can be implemented easily in MuRoSimF using the modeling concepts presented in the previous chapter. To implement a recursive algorithm, a set of ComputationObjects are defined, each one describing the respective calculations for one of the structural elements used to model the robot. The information exchanged between steps of the calculation are stored in algorithm dependent DataExchangeObjects. For each element of the robot's structure a ComputationObject is created during algorithm setup and connected to the object using an appropriate view. CalculationObjects connected to adjacent elements of the robot's structure are connected by an appropriate DataExchangeObject. If properties of structural elements are equal all the time, during setup of the algorithm the respective value's are shared by the objects using *explicit equality* (see Section 4.1.2). An example for an MBS connected to a specific algorithm is shown in Figure 5.1.

To perform a calculation, the CalculationObjects have to be called in the proper order. For recursive algorithms this order is obtained by visiting all nodes of the model tree depth first and storing the connected CalculationObjects in this order. If a calculation starts at the base of the model, this list is iterated beginning from the front. For algorithms starting at the leaves, the list is iterated backwards.

To define any algorithm, the algorithm specific DataExchangeObject and one CalculationObject for each type of structural element have to be provided. It is possible to have more than one CalculationObject for the same type of structural element to provide specialized (and thus simplified) calculations for special cases of the structural type (e. g. for a joint rotating around the x-axis being a special case of a revolute joint).

To facilitate connecting a basic algorithm to a simulation model, all necessary elements for an algorithm are aggregate into one class. This class provides a factory which is capable of creating the necessary CalculationObjects and DataExchangeObjects. When connecting a model to an object of the respective algorithm's class, the tree-structure of the model will be iterated

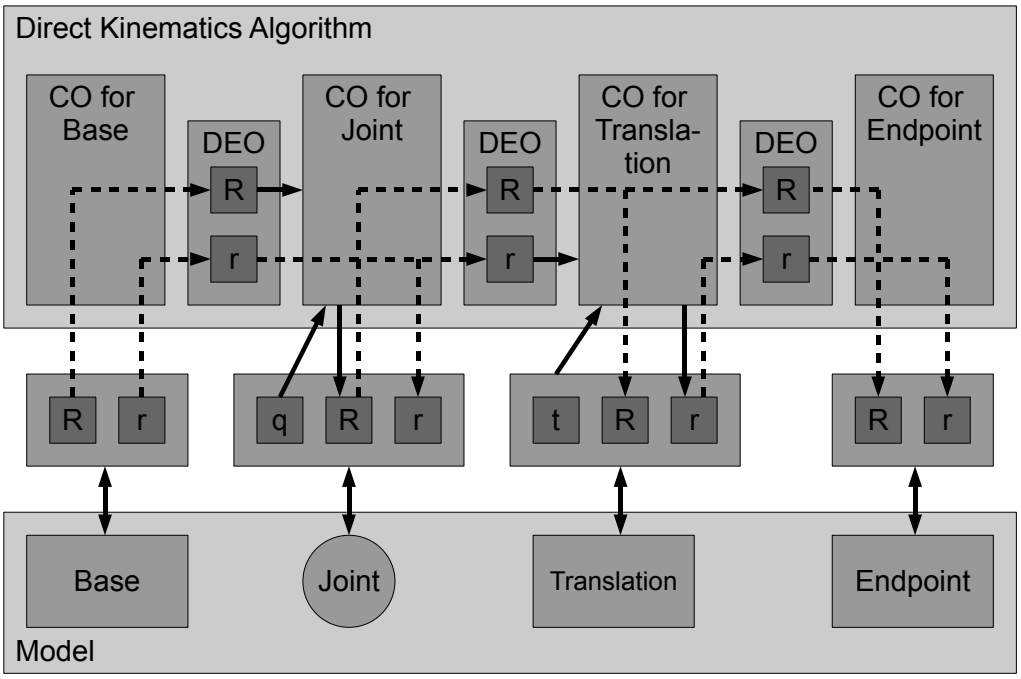


Figure 5.1: Calculation of direct kinematics: An algorithm calculating the forward kinematics of a kinematic chain is attached to the model representation of the robot. For each structural element a `ComputationObject` (CO) is created during algorithm setup and connected to respective element by a view. The `ComputationObjects` are connected by `DataObjects` (DO) transferring the relevant data between the COs. Arrows describe the data-flow during calculation. Dashed arrows denote that values are always equal and that the respective value holders are setup to share the value during algorithm setup, thus avoiding repeated assignments during runtime. Note that the value named "t" in the view of the translation-object refers to the modeling variable ${}^{\mathrm{pred}(i)}\mathbf{r}_i$ (fixed translation).

depth first by the factory creating an appropriate `CalculationObject` for each element of the model's structure as well as a `DataExchangeObject` between each pair of `CalculationObjects` connected to adjacent elements of the structure.

Direct Kinematics

To calculate the direct kinematics of a tree shaped kinematic structure `ComputationObjects` performing the calculations defined in Table 5.2 are connected to each element of the kinematic structure. The calculations are performed depth first beginning with the base of the structure. All properties calculated by a `ComputationObject` are used as input for the succeeding object, see Table 5.1. The formulae follow the standard procedure presented in many textbooks (e. g. [45, 103]), but are specialized to the seven modeling elements used in this thesis.

Table 5.1: Properties calculated by direct kinematics for each element i. These values are also used as input for the calculation for any object j with $i = \text{pred}(j)$.

Property	Symbol
Position	$\mathbf{r_i}$
Orientation	$\mathbf{R_i}$
Velocity	$\mathbf{v}_i, \omega_i$
Acceleration	$\dot{\mathbf{v}}_i, \dot{\omega}_i$

Inverse Kinematics

Inverse kinematics - the calculation of the joint angles and velocities required for a specific motion, has not yet been implemented, but can be added easily. Depending on the kind of robot, two methods would be obvious choices. If a specialized simulation is to be created for a robot with a known kinematic structure and a closed form solution for this specific structure exists, a specialized solver for this robot can be added to the simulation. If the robot's structure is not known a-priori or no closed form solution can be found, it is still possible to use a generic solver. This solver can be implemented using the Jacobian matrix of the robot, which can be calculated using the present methods for the calculation of the direct kinematics.

Inverse Dynamics

To calculate the inverse dynamics of a tree shaped kinematic structure the well known Recursive-Newton-Euler-Algorithm (RNEA, see Section 2.2.1 and [53]) has been implemented. This algorithm uses two sweeps over the kinematic structure of the robot. In the first sweep (going depth first from the base), direct kinematics is used to calculate the positions, velocities and accelerations of the robot for the desired motion. In a second sweep (going inwards from the endpoints), the necessary forces are calculated (see Table 5.3). The formulae of the second sweep adapted for the modeling elements present in `MuRoSimF` are given in Table 5.4.

The basic RNEA has been extended to use a viscous friction model (see [65]) and to cover inertial effects of the gears and the rotor of the joints drive (see [52]). Note that further extensions can easily be added to the algorithm by providing more complex `CalculationObjects` for the joints, which only will be used if additional parameters are present.

Forward Dynamics

Several different algorithms exist to calculate the forward dynamics of a robot. Which of these algorithms performs best depends on the structure of the robot.

Currently a basic and an extended implementation of the CRBA (see Section 2.2.1, cf. [146]) are provided [119]. Both implementations make use of the previously described implementation of the RNEA. The basic implementation covers the solution of the forward dynamics problem for robots with a fixed base, the extended version is able to handle robots with a free base. To handle the free base, the robot is expanded with a virtual 6DOF joint (allowing translation along and rotation around all axes). This joint is only used during the dynamics calculation

Table 5.2: Calculation of direct kinematics for the modeling elements used in `MuRoSimF`. Equations labeled with $(\star)$ are not calculated every timestep but instead set as explicitly equal during algorithm setup.

Element		
Base	No properties are calculated for the base. They may be set during model setup or by other algorithms.	
Fixed translation	$\mathbf{r}_i = \mathbf{r}_{\text{pred}(i)} + R_{\text{pred}(i)} \cdot^{\text{pred}(i)} \mathbf{r}_i$ $\mathbf{v}_i = \mathbf{v}_{\text{pred}(i)} + \omega_i \times^{\text{pred}(i)} \mathbf{r}_i$ $\dot{\mathbf{v}}_i = \dot{\mathbf{v}}_{\text{pred}(i)} + \dot{\omega}_i \times^{\text{pred}(i)} \mathbf{r}_i + \omega_i \times (\omega_i \times^{\text{pred}(i)} \mathbf{r}_i)$	$R_i = R_{\text{pred}(i)}\ (\star)$ $\omega_i = \omega_{\text{pred}(i)}\ (\star)$ $\dot{\omega}_i = \dot{\omega}_{\text{pred}(i)}\ (\star)$
Fixed rotation	$\mathbf{r}_i = \mathbf{r}_{\text{pred}(i)}\ (\star)$ $\mathbf{v}_i =^{\text{pred}(i)} R_i^{-1} \cdot \mathbf{v}_{\text{pred}(i)}$ $\dot{\mathbf{v}}_i =^{\text{pred}(i)} R_i^{-1} \cdot \dot{\mathbf{v}}_{\text{pred}(i)}$	$R_i = R_{\text{pred}(i)} \cdot^{\text{pred}(i)} R_i$ $\omega_i =^{\text{pred}(i)} R_i^{-1} \cdot \omega_{\text{pred}(i)}$ $\dot{\omega}_i =^{\text{pred}(i)} R_i^{-1} \cdot \dot{\omega}_{\text{pred}(i)}$
Variable translation	$\mathbf{r}_i = \mathbf{r}_{\text{pred}(i)} + R_{\text{pred}(i)} \cdot \mathbf{e}_{\mathbf{t}i} \cdot q_i$ $\mathbf{v}_i = \mathbf{v}_{\text{pred}(i)} + \omega_i \times \mathbf{e}_{\mathbf{t}i} \cdot q_i + \cdot \mathbf{e}_{\mathbf{t}i} \cdot \dot{q}_i$ $\dot{\mathbf{v}}_i = \dot{\mathbf{v}}_{\text{pred}(i)} + \dot{\omega}_i \times^{\text{pred}(i)} \mathbf{r}_i + \omega_i \times (\omega_i \times^{\text{pred}(i)} \mathbf{r}_i) + 2 \cdot \omega_i \times \mathbf{e}_{\mathbf{t}i} \cdot \dot{q}_i + \mathbf{e}_{\mathbf{t}i} \cdot \ddot{q}_i$	$R_i = R_{\text{pred}(i)}\ (\star)$ $\omega_i = \omega_{\text{pred}(i)}\ (\star)$ $\dot{\omega}_i = \dot{\omega}_{\text{pred}(i)}\ (\star)$
Variable rotation	$\mathbf{r}_i = \mathbf{r}_{\text{pred}(i)}\ (\star)$ $\mathbf{v}_i =^{\text{pred}(i)} R_i^{-1} \cdot \mathbf{v}_{\text{pred}(i)}$ $\dot{\mathbf{v}}_i =^{\text{pred}(i)} R_i^{-1} \cdot \dot{\mathbf{v}}_{\text{pred}(i)}$	$R_i = R_{\text{pred}(i)} \cdot R(\mathbf{e}_{\mathbf{r}i}, q_i)$ $\omega_i =^{\text{pred}(i)} R_i^{-1} \cdot \omega_{\text{pred}(i)} + \mathbf{e}_{\mathbf{r}i} \cdot \dot{q}_i$ $\dot{\omega}_i =^{\text{pred}(i)} R_i^{-1} \cdot (\dot{\omega}_{\text{pred}(i)} + \omega_{\text{pred}(i)} \times \mathbf{e}_{\mathbf{r}i} \cdot \dot{q}_i) + \mathbf{e}_{\mathbf{r}i} \cdot \ddot{q}_i$
Fork	No calculations required.	
End-point	No calculations required.	

Table 5.3: Properties calculated by inverse dynamics. Note that only $\mathbf{f}_i$ and $\mathbf{n}_i$ are exchanged for adjacent objects. As all elements except points are defined to massless $\mathbf{F}_i$ and $\mathbf{N}_i$ only need to be calculated for these. The joint force τ_i only has to be calculated for joints.

Property	Symbol
Force and torque an element needs to experience by its predecessors	$\mathbf{f}_i$, $\mathbf{n}_i$
Force and torque an element needs to experience on center of mass	$\mathbf{F}_i$, $\mathbf{N}_i$
Force resp. torque to be exerted by a joint	τ_i

Table 5.4: Calculation of inverse dynamics by the RNEA for the modeling elements used in `MuRoSimF`. Note that the successor $\mathrm{succ}(i)$ of a link i only is defined for links with exactly one successor (that is all but fork and point). Note further that the force $\mathbf{f}_i$ and the torque $\mathbf{n}_i$ which a link must experience by its predecessor is represented in the predecessor's coordinate frame.

Element		
Fixed base	$\mathbf{f}_i = \mathbf{f}_{\mathrm{succ}(i)}$	$\mathbf{n}_i = \mathbf{n}_{\mathrm{succ}(i)}$
Fixed translation	$\mathbf{f}_i = \mathbf{f}_{\mathrm{succ}(i)}$	$\mathbf{n}_i = \mathbf{n}_{\mathrm{succ}(i)} - \mathbf{f}_{\mathrm{succ}(i)} \times {}^{\mathrm{pred}(i)}\mathbf{r}_i$
Fixed rotation	$\mathbf{f}_i = {}^{\mathrm{pred}(i)}R_i \cdot \mathbf{f}_{\mathrm{succ}(i)}$	$\mathbf{n}_i = {}^{\mathrm{pred}(i)}R_i \cdot \mathbf{n}_{\mathrm{succ}(i)}$
Variable translation	$\mathbf{f}_i = \mathbf{f}_{\mathrm{succ}(i)}$ $\tau_i = \mathbf{e}_{\mathbf{t}i} \cdot \mathbf{f}_i + \mu_{vi} \cdot \dot{q} + \rho_i^2 \cdot \ddot{q}_i$	$\mathbf{n}_i = \mathbf{n}_{\mathrm{succ}(i)} - \mathbf{f}_{\mathrm{succ}(i)} \times \mathbf{e}_{\mathbf{t}i} \cdot q_i$
Variable rotation	$\mathbf{f}_i = R(\mathbf{e}_{\mathbf{r}i}, q_i) \cdot \mathbf{f}_{\mathrm{succ}(i)}$ $\tau_i = \mathbf{e}_{\mathbf{r}i} \cdot \mathbf{n}_i + \mu_{vi} \cdot \dot{q} + \rho_i^2 \cdot \ddot{q}_i$	$\mathbf{n}_i = R(\mathbf{e}_{\mathbf{r}i}, q_i) \cdot \mathbf{n}_{\mathrm{succ}(i)}$
Fork	$\mathbf{f}_i = \sum_{i=\mathrm{pred}(j)} \mathbf{f}_j$	$\mathbf{n}_i = \sum_{\mathrm{pred}(j)} \mathbf{n}_j$
Endpoint	$\mathbf{F}_i = m_i \cdot \dot{\mathbf{v}}_i$ $\mathbf{f}_i = \mathbf{F}_i - \mathbf{f}_{\mathbf{ext},i}$	$\mathbf{N}_i = I_i \cdot \dot{\omega}_i + \omega_i \times I_i \cdot \omega_i$ $\mathbf{n}_i = \mathbf{N}_i - \mathbf{n}_{\mathbf{ext},i}$

and the solution is mapped to the linear and angular acceleration of the robot's base after the calculation.

The CRBA relies on the same modeling data as the RNEA. During calculation it requires information on the position q, the velocity $\dot{q}$ and the force τ of each joint of the robot. The extended version requires additional information on the position $\mathbf{r}$, orientation R, linear velocity $\mathbf{v}$ and angular velocity ω of the base.

This implementation can be transparently replaced by an implementation of the Articulated Body Algorithm, which is known to be of better performance for robots with more than 9 DOF (see Section 2.2.1).

Numerical Integration

Numerical integration is provided by specific `Integrators` which can be attached to the robot's base and joints. An `Integrator` will read the respective object's velocity and acceleration and calculate the resulting position and velocity for a given timestep. Currently `Integrators` providing Euler's method are implemented. `Integrators` for more complex explicit (e. g. Runge-Kutta) methods can be added.

Adaptability and Extendability

The methodologies used for modeling the MBSs and for structuring the MBS algorithms allow for an easy adaption of the algorithm to specific needs. In the basic algorithms described before, only seven distinct modeling elements are used with one `CalculationObject` for each type of element and algorithm. To cover further effects, e. g. elasticities of gears, the algorithms can be extended by two steps:

1. New structural elements must be included.
2. New `CalculationObjects` capable of handling the new structural elements must be added to the algorithms.

Due to the modular structure of the models and the algorithms, no changes need to be made to existing structural elements or `CalculationObjects`.

5.1.2 Specialized Algorithms

Specialized algorithms for robot motion simulation are algorithms which are limited to robots with a specialized structure. The algorithms may make further assumptions on the kind of motion or the environment. Exploiting the knowledge of the structure and the other assumptions made, these algorithms are simpler in structure and have lower computational complexity than general algorithms, thus allowing the creation of more efficient simulations.

Kinematical Simulation of Vehicles with Differential Drive

When assuming motion in a plane and neglecting slipping, the velocity of a vehicle with differential drive is fully determined by the structure of the vehicle (described by the distance w_{dist}

and radius w_{rad} of the wheels) as well as the velocity of the wheels. As described in many textbooks (e. g. [46]) the velocity of the robot's base can be calculated by

$$v_l = \dot{q}_l \cdot w_{rad} \tag{5.1}$$

$$v_r = \dot{q}_r \cdot w_{rad} \tag{5.2}$$

$$\mathbf{v}_b = \begin{pmatrix} \frac{v_l + v_r}{2} \\ 0 \\ 0 \end{pmatrix} \tag{5.3}$$

$$\omega_b = \begin{pmatrix} 0 \\ 0 \\ \frac{v_r - v_l}{w_{dist}} \end{pmatrix}. \tag{5.4}$$

The implementation of this method makes use of a view, that only provides access to the relevant properties of the base and the drives of the wheels as well as the information on the structure of the vehicle. All calculations are performed using this view, thus decoupling the simulation method from the model representation of the robot.

Kinematical Simulation of Biped Robots

Using the calculation of the robot's direct kinematics, an efficient algorithm for biped walking simulation can be derived. The basic algorithm discussed in the following section makes two assumptions:

1. The robot's motion is limited to a plane.
2. The standing foot stands firmly on the plane of motion without slipping or shaking.

The parts of the algorithms beyond the direct kinematics calculation only regard three distinguished objects of the robot, namely the base, the left foot and the right foot (see Section 4.3). Thus for the implementation of the following methods, a view is used which provides only access to these three objects, decoupling the implementation from the model. For each of these objects reading and writing access to the properties position and orientation are required. In the basic variation, no further properties are affected by the algorithm. The advanced versions of this method require access to more properties of the robot's base and feet, but do not require access to more objects.

Basic Algorithm

In each time step of the simulation, calculation of the robot's direct kinematics is used to determine position and orientation of both feet of the robot. After this it is checked whether the standing foot from the last time step can be kept, or if the standing foot has to be exchanged. In case of an exchange of the standing foot, position and orientation of the new standing foot in the plane of motion are calculated. Based on the standing foot position and orientation of the robot's base are corrected and finally the direct kinematics is calculated again for all elements of the robot (see Figure 5.2).

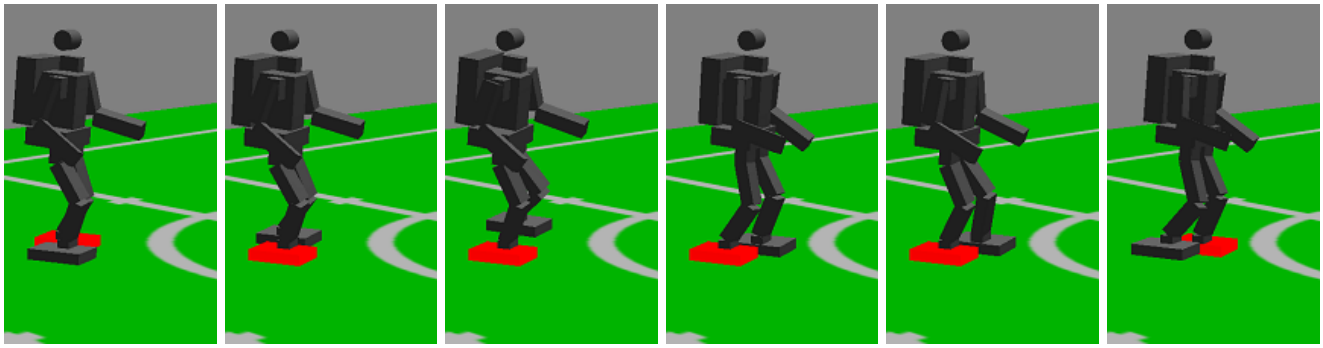


Figure 5.2: Kinematical walking simulation of a humanoid robot. The standing foot is colored red in the images [62].

The algorithm can be summarized formally as follows[1]:

1. Use direct kinematics to calculate $\mathbf{r}_l$, R_l, $\mathbf{r}_r$, R_r
2. Check whether standing foot $s = l$ or $s = r$
3. Calculate position and orientation of the base relatively to the standing foot:

$$ {}^{sf}R_b = R_s^{-1} \cdot R_b \tag{5.5} $$
$$ {}^{sf}\mathbf{r}_b = R_s^{-1} \cdot (\mathbf{r}_s - \mathbf{r}_b) \tag{5.6} $$

4. Calculate corrected position $\mathbf{r}_s$ and orientation R_s of the standing foot:
   a) If the standing foot hasn't changed, use the respective values from the last time step.
   b) If the standing foot has changed, correct the values, so that the standing foot is in the plane of motion.
5. Adjust position and orientation of the base:

$$ R_b = R_s \cdot {}^{s}R_b \tag{5.7} $$
$$ \mathbf{r}_b = \mathbf{r}_s + R_s \cdot {}^{s}\mathbf{r}_b \tag{5.8} $$

Calculation of the Standing Foot

The calculation of the standing foot is based on the assumption made before, namely that it stands firmly on the ground. Thus the other foot must be above the plane defined by the standing foot. To determine the standing foot, the position of each foot is compared to the standing plane of the respective other foot. Three cases may occur (see Figure 5.3):

- Exactly one foot is above the plane defined by the other foot: this foot is the standing foot to use for the following calculation. If it hasn't been the standing foot before, position and orientation have to be corrected.
- Each foot is above the plane defined by the other foot: Both feet are candidates for the standing foot. Following the assumption made before, the foot which has been the standing foot before is used furthermore.

[1] Note that the indices l, r and s denote the left foot, right foot and current standing foot of the robot. The index b denotes the robot's base.

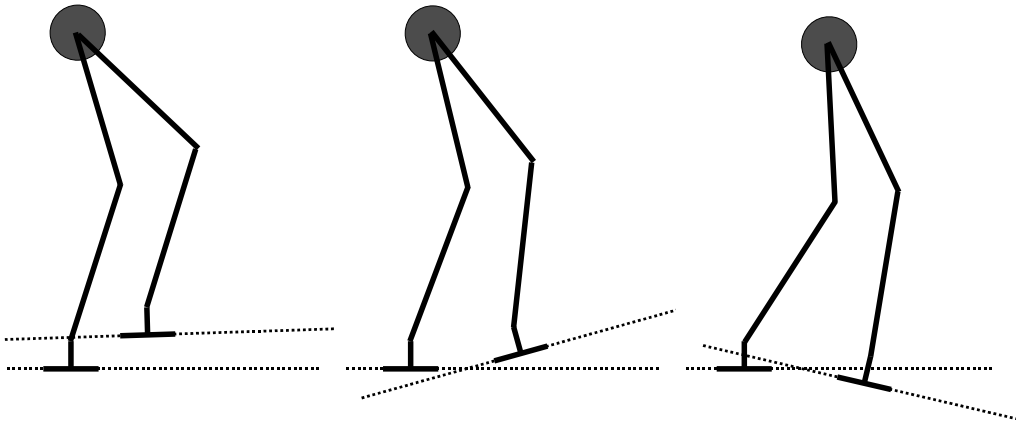


Figure 5.3: Calculation of the standing foot. Three different cases must be considered: Left: one unique standing foot. Middle: two possible standing feet. Right: no possible standing foot, as either foot is below the standing plane of the other.

- Neither of the feet are above the plane defined by the other foot: This case cannot occur during normal walking motion. It contradicts the algorithm's assumption and thus cannot be handled properly.

Extending the Algorithm to Cover Velocity and Acceleration

If the simulation model of the robot's servo-motors provides information on the velocity $\dot{q}$ and the acceleration $\ddot{q}$ of the robot's joints, the algorithm described in the last section can be extended to the calculation of the robots velocity and acceleration. The basic structure of the algorithm is not changed, but only extended to cover the velocity ($\mathbf{v}$, ω) and acceleration ($\dot{\mathbf{v}}$, $\dot{\omega}$) of the robot's feet and base. This leads to the following extensions:

- Additional assumptions demanding that velocity and acceleration of the standing foot are zero are made: $\mathbf{v}_{sf} = \mathbf{0}$, $\omega_{sf} = \mathbf{0}$, $\dot{\mathbf{v}}_{sf} = \mathbf{0}$ and $\dot{\omega}_{sf} = \mathbf{0}$

- The calculation of the direct kinematics must cover velocity as well as acceleration. To enable an easy calculation of the relative velocity and acceleration of the foot, the respective values of the base are set to zero: $\mathbf{v}_b = \mathbf{0}$, $\omega_b = \mathbf{0}$, $\dot{\mathbf{v}}_b = \mathbf{0}$ and $\dot{\omega}_b = \mathbf{0}$. Calculation of the direct kinematics than yields the relative velocity and acceleration of the feet (with respect to the base) in world coordinates.

- In the third step of the algorithm, the velocity and acceleration of the standing foot relative to the base must be calculated added:

$$
\begin{aligned}
{}^{b}\omega_s &= R_b^{-1} \cdot (R_s \cdot \omega_s - R_b \cdot \omega_b) && (5.9)\\
{}^{b}\mathbf{v}_s &= R_b^{-1} \cdot (R_s \cdot \mathbf{v}_s - R_b \cdot \mathbf{v}_b) && (5.10)\\
{}^{b}\dot{\omega}_s &= R_b^{-1} \cdot (R_s \cdot \dot{\omega}_s - R_b \cdot \dot{\omega}_b) && (5.11)\\
{}^{b}\dot{\mathbf{v}}_s &= R_b^{-1} \cdot (R_s \cdot \dot{\mathbf{v}}_s - R_b \cdot \dot{\mathbf{v}}_b) && (5.12)
\end{aligned}
$$

- In the fifth step of the algorithm, correction of velocity and acceleration of the base must be added:

$$\begin{aligned}
\omega_b &= -{}^b\omega_s && (5.13)\\
\mathbf{v}_b &= -{}^b\mathbf{v}_s - \omega_b \times {}^b\mathbf{r}_s && (5.14)\\
\dot{\omega}_b &= -{}^b\dot{\omega}_s && (5.15)\\
\dot{\mathbf{v}}_b &= -{}^b\dot{\mathbf{v}}_s - \dot{\omega}_b \times {}^b\mathbf{r}_s - \omega_b \times (\omega_b \times {}^b\mathbf{r}_s) - \omega_b \times {}^b\mathbf{v}_s && (5.16)
\end{aligned}$$

Extending the Algorithm to Cover Slipping

A biped robot's motion often suffers from slipping. The presented kinematic simulation method can be extended to cover this effect in a simplified way. To do this, the position of the standing foot no longer is kept fixed. Instead, the foot is allowed to move to some extent depending on the relative velocity of the foot calculated by the direct kinematics. Currently the slipping motion is proportional to the relative velocity of the foot, but more complicated models (e. g. depending on the velocity of the foot or considering the phase of the motion) can be added if need arises.

Applications

The algorithm described in this section makes several simplifying assumptions concerning the robot's motion. On the one hand there are the explicit assumptions on the behavior of the standing foot, namely that it does not move as long as the robot is standing on this foot. This assumption inhibits the simulation of any motion beyond walking (e. g. falling down). On the other hand the purely kinematical consideration of the robot's joints ignores effects like elasticity or overloading of the servo-motors. Thus the algorithm is not able to simulate inner motions of the robot perfectly.

Nevertheless the algorithm is suitable for many application. If a precise physical simulation of the robot's motion is not required, the simplifying assumptions will lead to several advantages [62]: The low computational complexity of the algorithm allows for the simulation of many robots, allowing SIL-testing of the behavior of teams of robots in real-time. Due to the fixed standing foot and the simplified model of the joints, several undesired motions of the real robot are not simulated, thus ruling out these effects as cause for observed errors. This enables extensive tests of the robot's behavior without masking errors in the control software by mechanical effects of the real hardware.

5.1.3 General Algorithms

General algorithms for motion simulation impose fewer restrictions on the structure of the simulated robot and make fewer assumptions on the environment. Several algorithms are provided in `MuRoSimF` on different levels of physical detail.

Kinematical Motion of a Single Point

Motion of a system described as a single point can be simulated by setting either the position and orientation, the velocity or the acceleration. If position and orientation are set directly, no

further calculations are needed. In case of velocity or acceleration, an additional integration step hast to follow.

If the motion of the base of a robot is simulated this way, the motion of other elements of the robot still may be simulated calculating their relative position, velocity and acceleration using direct kinematics. Obviously this will neglect any effects of the motion of the additional robot links on the motion of the base. Nevertheless this approach may be used if contact of the links can be neglected and a sufficient model for the motion of the base exists.

Dynamical Motion of a Single Point

Like the kinematical motion simulation for a single point, the dynamical motion simulation for a single point reduces a system to one point in space and does not consider any additional parts of a robot. Unlike the kinematical simulation though, it calculates the acceleration of the single point by solving the dynamics equations 2.1 and 2.2. Further calculation is handled similar to the kinematic simulation by integration. Again, additional relative motion of further links of a robot can be simulated by direct kinematics, but again this motion will not affect the calculation of the motion of the base.

Simplified Dynamics

To consider the effects of contacts of the robot's moving parts with the environment, a simplified dynamics simulation is provided. This simulation method considers the motion of the robot's links only on kinematical level. Contact forces derived from the collision detection are propagated to the center of mass of the robot, where the equations of dynamical motion are solved.

Algorithm

1. Calculate spatial arrangement of the robot's bodies using direct kinematics.
2. Calculate total mass of the system
$$m_{com} = \sum_{i \in bodies} m_i \tag{5.17}$$
3. Calculate common center of mass:
$$\mathbf{r}_{com} = \frac{1}{m_{com}} \sum_{i \in bodies} m_i \mathbf{r}_i \tag{5.18}$$
4. Calculate common inertia tensor of the system at the common center of mass.
5. Calculate force and torque acting at the common center of mass resulting from all external forces:
$$\mathbf{f}_{com} = \sum_{i \in bodies} R_i \cdot \mathbf{f}_i \tag{5.19}$$
$$\mathbf{n}_{com} = \sum_{i \in bodies} R_i \cdot \mathbf{n}_i + (\mathbf{r}_{com} - \mathbf{r}_i) \times (R_i \cdot \mathbf{f}_i) \tag{5.20}$$
6. Calculated acceleration of the common center of mass by solving 2.1 and 2.2.

Table 5.5: Overview of methods for motion simulation. The methods are sorted by rising accuracy.

Method	Category	Simulation of base motion	Simulation of joint motion
Point kinematics	general	kinematical	not simulated
Point dynamics	general	dynamical	not simulated
Biped motion simulation	specialized	kinematical	kinematical
Differential drive simulation	specialized	kinematical	kinematical
Simplified dynamics	general	dynamicsl	kinematical
Full forward dynamics	general	dynamical	dynamical

Applications

The simulation method has been used successfully for several robots with different structure including biped robots [61, 62], quadruped robots [63] and wheeled vehicles [60]. As the method only uses a kinematical simulation for the joint, no effects of external forces on the joints can be considered. Nevertheless a wide variety of motions can be simulated without limitations on the robot's structure or the environment.

Full MBS Dynamics

The full MBS dynamics of a robot can be simulated with the forward dynamics methods presented in Section 5.1.1.

5.1.4 Scalability and Adaptability of Motion Simulation

The algorithms presented in the previous section allow the simulation of a robot's motion on several levels of detail. Basic simulation methods for the robot's motion within its environment (neglecting the inner motion of the robot) are provided on kinematic and dynamic level. Further on methods are described which additionally cover the motion of the robot's joints.

The simulation methods can be categorized into two groups: Specialized and general methods. General methods are not dependent on the structure of the simulated robot. They are provided on a very abstract level only considering the motion of the base and on two levels considering the robots dynamics (general forward dynamics and simplified dynamics). Additionally, several specialized methods have been presented. All of these rely on a specific structure of the robot and simulate the motion of the robot on a kinematic level. An overview of the presented methods if given in Table 5.5.

As each of the presented methods only considers the motion of one robot, it is possible to combine these methods within one simulation, thus allowing the simulation of several robots on different levels of accuracy within the same simulation. Currently all provided methods for dynamics simulation are based on algorithms using generalized coordinates. Other algorithm can be implemented likewise and be connected with the simulation models. As discussed before, the basic MBS algorithms can be extended easily to cover further structural elements.

5.2 Collision Detection and Handling

If physical interaction of a robot with its environment has to be investigated, a simulation must be able to detect collisions and handle their effects. Both tasks, detecting and handling collisions, can be solved in several ways differing in physical accuracy and computational complexity. Collision detection can be subdivided further in the task of detecting collisions between a pair of objects and the task of selecting which pairs of objects to test for collisions. For an adaptable simulation it is desirable to select the algorithms to use for each of these three subtasks independently.

In `MuRoSimF` the three subtasks are handled by different classes. Selection of the pairs of objects to check for collisions is done by a subclass of `CollisionDetectionTask`. The strategies used for this are discussed in Section 5.2.1.

Independent of the selection process, for each pair of objects a subclass of `CollisionDetector` is used to detect collisions. If collisions are found, the collisions are handled by a subclass of `CollisionHandler`. The precise subclass to instantiate for each of the three subtasks can be determined independently for each simulation.

To enable the flexible exchange of the subtasks, the view concept described in Section 4.2 is used. Basic access to the objects in the simulation is given to the subtasks by the `CollisionDetectionView` and `CollisionHandlingView`. As different approaches for these subtasks call for different data, each subclass of `CollisionDetector` and `CollisionHandler` may use a specialized subclass of the respective view. Besides methods for their respective task, each of these classes provides the functionality to create their respective view for objects.

During setup of the algorithm, the concrete `CollisionDetector` and `CollisionHandler` to use are registered with the `CollisionDetectionTask`. After this, the `CollisionDetectionTask` will use these to create their respective views for each object in the scene. During runtime of the simulation, after selecting a pair of objects for collision detection, the `CollisionDetectionTask` will call the used `CollisionDetector` with the `CollisionDetectionView` of each object. If collisions are found, a list of the collisions and the `CollisionHandlingView` is given to the `CollisionHandler` in use.

5.2.1 Selection of Object Pairs

The first step of collision detection is to determine for which pairs of objects in a simulated scene the collision detection has to be performed. This decision is made by the `CollisionDetectionTask` in two sub steps.

In the first sub step, it is determined which pairs of compound objects need further investigation. To guide this process, for each compound registered for collision detection it is stored, if the compound needs checking for inner collisions and if the compound is static. No collision tests between two static compounds are performed.

In the second sub step, for a pair of compound objects, pairs of objects are determined which need further examination. The easiest way to do this is to check each object of the first compound for collisions with each object of the second, leading to an $O(n^2)$ runtime. To improve this, hierarchies of bounding volumes are used (see [115, 62]).

Using such hierarchies allows for two mayor design decisions: the kind of hierarchy and the kind of bounding volume to use. In the present implementation, both can be chosen independently.

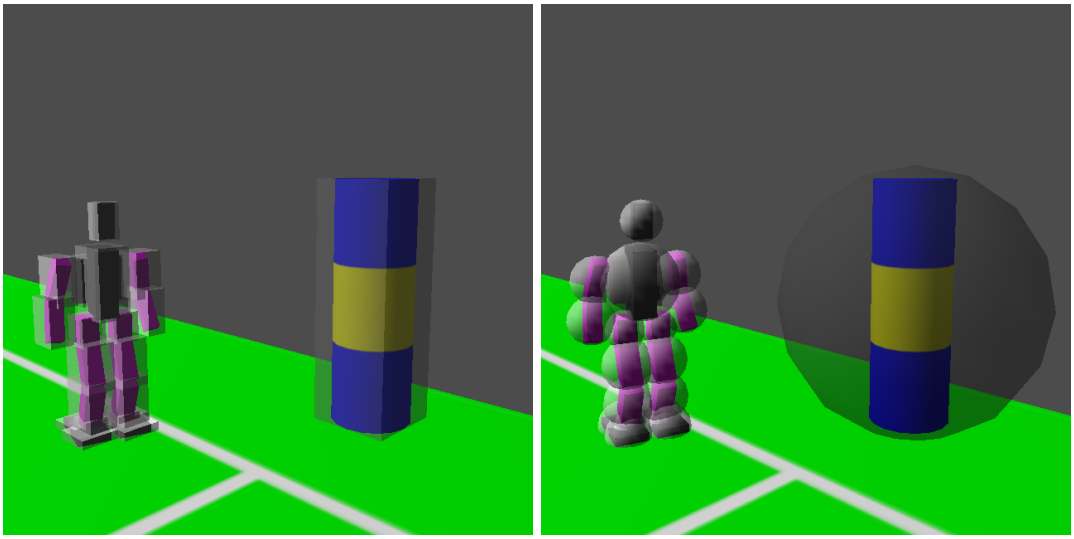


Figure 5.4: Different bounding volumes used for collision detection pre testing. The images show the same scene from the humanoid robot simulation, on the left using axis aligned bounding boxes, on the right using bounding spheres.

Currently two kinds of hierarchies are supported:

- A two level hierarchy storing one bounding volume for the whole compound object and one bounding volume for each body of the compound object. This hierarchy provides a faster building of the tree, but may lead to more tests on the object level.
- A binary tree of bounding volumes storing the bounding volume for the whole compound object in its root and the bounding volume for the single bodies in its leaves. This hierarchy needs less tests on object level, but has a slower building of the tree.

The hierarchies are realized as template classes allowing to choose the kind of bounding volume and an appropriate view as a template parameter. By this, it is possible to make a tradeoff between bounding volumes with low computational requirements but poor fitting (e. g. spheres) or better fitting, but more expensive bounding volumes (e. g. bounding boxes aligned to the world coordinate frame), see Figure 5.4.

5.2.2 Collision Detection

The task of any `CollisionDetector` is to find collisions between a pair of objects. Within `MuRoSimF` a collision is described in the coordinate systems of both colliding objects. The data used to describe a collision are the position of the collision, the penetration depth and the normal vector of the collision, see Table 5.6.

Objects are accessed through a specialized view by the `CollisionDetector`. The basic `CollisionDetectionView` provides read-only access to the position and orientation of the object and provides a method to add collisions to the object. Any further properties needed by a specialized `CollisionDetector` must be added in their respective derived view.

For general use, a `CollisionDetector` for primitive shapes (box, sphere, capsule, cylinder and plane) has been implemented following the principles laid out in [49]. The specialized

Table 5.6: Description of a collision.

Property	Symbol	Type
Collision-position	$\mathbf{p}_{col}$	$\mathbb{R}^3$
Collision-normal	$\mathbf{n}_{col}$	$\mathbb{R}^3$
Penetration depth	d_{col}	$\mathbb{R}$

`CollisionDetectionView` utilized by this detector provides additional access to the properties of the object describing type and dimensions of the object's shape.

If different kinds of shapes are to be used in a simulation, a new `CollisionDetector` with an appropriate `CollisionDetectionView` must be derived that is able to handle such shapes.

5.2.3 Collision Handling

The subtask of handling collisions is implemented independently from the subtask of detecting collisions. Whenever collisions between a pair of objects are detected a `CollisionHandler` is called with a list of the detected collisions. The `CollisionHandler` then uses its own `CollisionHandlingView` to manipulate the properties of the colliding objects according to the collision handling algorithm in use.

Currently a `CollisionHandler` using a soft collision model has been implemented, which is discussed in the next paragraph. After this, other approaches and their possible implementation within the presented framework are discussed.

Soft Contact Model

A soft contact model is a contact model which allows the penetration of the colliding objects. The soft contact model implemented in `MuRoSimF` calculates forces and torques resulting from the collision. Two different effects are considered by the model:

- normal forces resulting from the collision of the objects pushing the objects apart
- frictional forces caused by relative motion of the objects.

All forces are calculated in the collision position $\mathbf{p}_{col}$ provided by the collision detection. Besides the information provided by the collision detection, the collision detection algorithm requires additional information (see Table 5.7) on the collision and the colliding objects.

To consider friction, the relative velocity $\mathbf{v}_{rel}$ in the contact point is required. This information is obtained by the collision handler using views on the colliding objects which provide information on each object's linear and angular velocity.

Material constants (see Table 5.8) describing the collision are required. These constants depend on the materials of both colliding objects. A table is maintained, recordig these constants for each pair of material possibly colliding in the respective simulation.

Table 5.7: Additional properties required by the soft collision handling algorithm from both colliding objects.

Property	Symbol
Position	$\mathbf{r}$
Orientation	R
linear Velocity	$\mathbf{v}$
angular Velocity	ω
Material	(see Table 5.8)

Table 5.8: Material constants describing the contact situation between two objects.

Effect	Property	Symbol
Collision	Rebound constant	$c_{rebound}$
	Damping constant	c_{damp}
	Bounciness	c_{bounce}
	Maximum penetration	$c_{col,max}$
Friction	Friction constant	μ

The calculation of the normal force resulting from the collision is based on a spring-and-damper-model depending on the penetration depth and the relative velocity of the colliding objects in the direction of the collision-normal:

$$v_{rel,normal} = \mathbf{v}_{rel} \cdot \mathbf{n}_{col} \tag{5.21}$$

$$f_{normal} = \min(d_{col}, c_{col,max}) \cdot c_{rebound} \tag{5.22}$$

$$\mathbf{f}_{col} = f_{normal} \cdot \mathbf{n}_{col} \cdot \begin{cases} 1, & \text{if } v_{rel,normal} \leq 0 \\ c_{rebound}, & \text{else} \end{cases} \tag{5.23}$$

$$\mathbf{f}_{damp} = v_{rel,normal} \cdot \mathbf{n}_{col} \cdot c_{damp}. \tag{5.24}$$

The calculation of the friction is based on a viscous model depending on the relative velocity of the of the colliding objects orthogonal to the collision normal:

$$\mathbf{v}_{rel,tangential} = \mathbf{v}_{rel} - v_{rel,normal} \cdot \mathbf{n}_{col} \tag{5.25}$$

$$\mathbf{f}_{fric} = \mathbf{v}_{rel,tangential} \cdot \mu. \tag{5.26}$$

The resulting force is applied to both colliding objects in their respective collision position.

Integration of Other Contact Models

The simulation is not limited to the soft contact model discussed in the previous section. The separation of collision detection and collision handling allows for the easy integration of other models by providing different collision-handlers. For collision handling methods which do not resolve the contact situation per collision, but require a global view (e. g. the methods based on the Lagrange-multiplier discussed in Section 2.2.1), an adapted collision handler can be used to store collisions information in a proper form for a later collision resolving stage.

5.2.4 Scalability and Adaptability of the Collision Detection Subsystem

The collision detection subsystem presented has the merit of being adaptable to very different requirements. An interface is defined which allows independent handling of the subtasks of selecting objects that require collision detection, performing the collision detection and handling detected collisions. For each of these subtasks the concrete method to use may be chosen independently of the other subtasks. Due to this flexibility it is possible to adjust performance and accuracy of the simulation by choosing more or less accurate (and thus more or less computationally complex) methods for the three subtasks. In Section 7.4.2 an application is discussed which utilizes a domain specific collision detection algorithm.

5.3 Visualization

Depending on the concrete use-case, requirements on the visualization for a simulation vary greatly.

5.3.1 General Concept

The visualization module of `MuRoSimF` has been developed to allow a flexible configuration of *what* is to be displayed and *how* it is to be displayed. To enable an easy exchange of the rendering system used to display the graphics, the rendering setup has been completely separated from the implementation of the rendering system.

Rendering Setup

A rendering setup describes, which properties are to be rendered for a given object of a scene. It is possible to select the properties for each object individually.

Further on, the rendering setup stores information about light sources to use during rendering.

Rendering System

Any rendering system consists of three types of classes, the `RenderManager`, the `ObjectRenderer` and the `PropertyRenderer` base class. To create a specific rendering system, subclasses of `RenderManager`, `ObjectRenderer` and `PropertyRenderer` must be derived. The basic interaction of these classes is defined as follows:

The `RenderManager` is the central instance to initialize setup of a rendering system as well as rendering. During setup the `RenderManager` will iterate through the rendering setup and create an appropriate instance `ObjectRenderer` for each object defined in the rendering setup. For rendering, the `RenderManager` base class will iterate through all `ObjectRenderers` created during setup and use each of these to render their individual object. Besides the methods used during setup and rendering, the `RenderManager` provides further methods for picking, interaction and capturing of images which must be implemented in the rendering system specific subclasses.

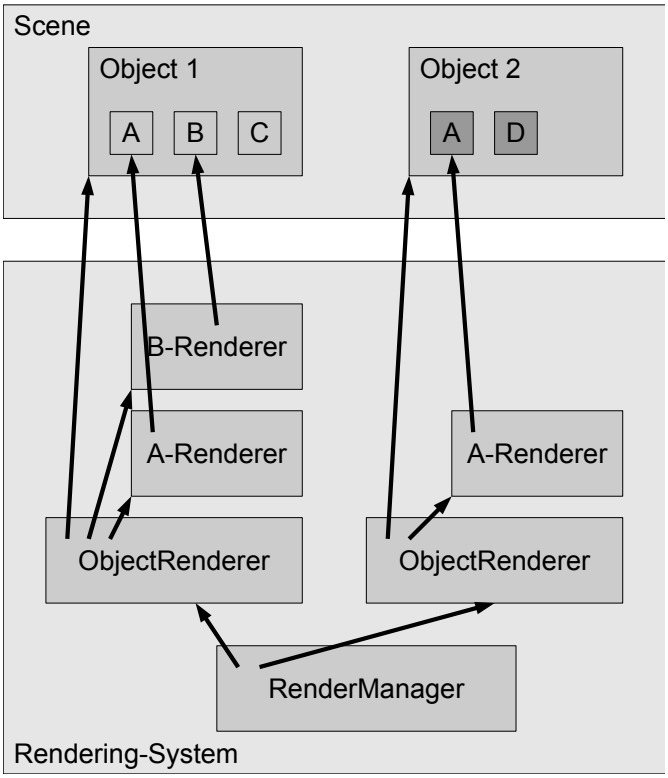


Figure 5.5: Structure of the rendering system. In this scene only the properties A and B of the objects are to be rendered. No renderers for other properties are created. If an object does not have a property, the specific renderer is not created for this object. Arrows within the rendering-system indicate creation of objects and calling directions. Arrows from the rendering system to the scene indicating read-only data access.

An `ObjectRenderer` is used to render the properties of one object of a scene. During setup, the `ObjectRenderer` tries to create an appropriate `PropertyRenderer` for each property defined in the rendering setup. During rendering the `ObjectRenderer` will iterate through all `PropertyRenderers` created before, using each to render one property of the object.

For each kind of property to be rendered, a specific subclass of `PropertyRenderer` must be derived. During setup, any `PropertyRenderer` will check, if the specific property is present in an object. Only if the property is present, the `PropertyRenderer` can be used later on.

The interaction of the basic classes of the rendering system is shown in Figure 5.5.

5.3.2 Implementations

A `RenderManager` with specific `ObjectRenderer` and a set of `PropertyRenderers` have been implemented for two different rendering systems.

To allow for realtime rendering, an implementation based on OpenGL ([133]) has been developed. Besides rendering of the simulated scene in realtime, this implementation provides easy user interaction with the scene by picking and dragging objects in the scene.

For the generation of high quality images (e. g. for visualization purposes) a second rendering system based on the POVRay-raytracing software [12] has been implemented. The rendering system generates POVRay-input files for each frame which is to be visualized. These input files

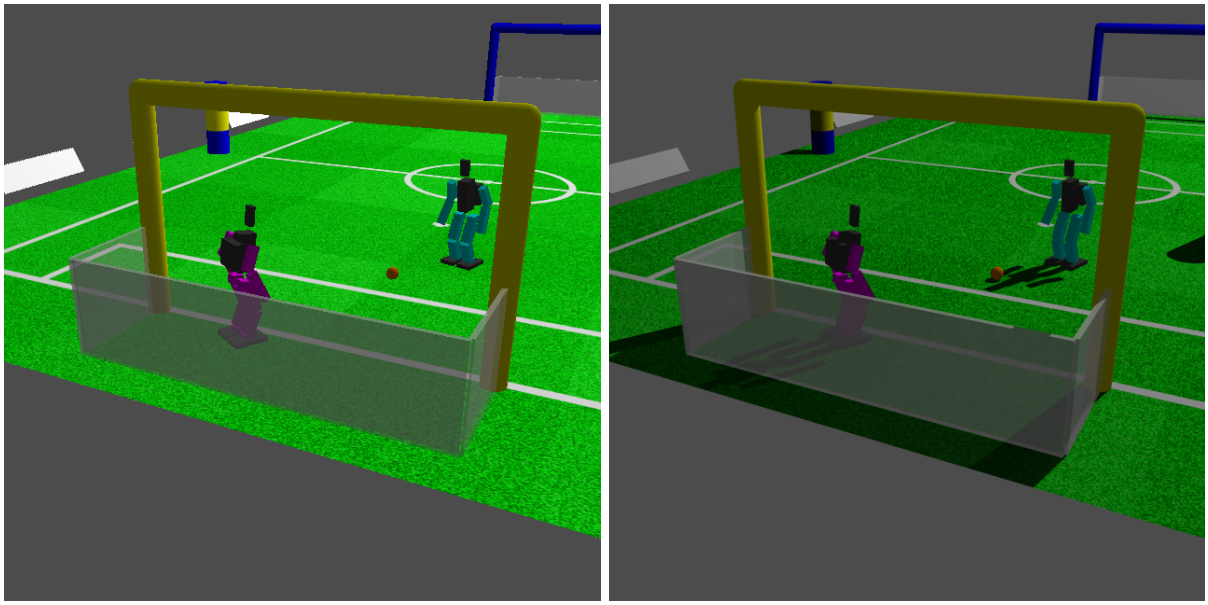


Figure 5.6: Example for different rendering systems. The images shown here are a case study for new goals to be introduced in the RoboCup humanoid league in 2010. Left: The scene rendered with the OpenGL real-time renderer. Right: The same scene rendered offline by POVRay. Note the additional shadows and the nicer transparency of the goals.

later on are processed offline to generate images. An example for the use of both rendering systems is given in Figure 5.6.

Both rendering systems provide a range of `PropertyRenderers` for different purposes. For simple visualization of the simulated scene, the shapes of the simulated objects (as meshes or as primitives) may be rendered (optionally with textures). This visualization can be augmented with other information, e. g. sensor readings or information on detected collisions by adding appropriate `PropertyRenderers`. Besides displaying the outward appearance of the simulated robots, it is also possible to visualize more technical information, e. g. the kinematic structure. Examples for different displays are given in Figure 5.7. Note that all rendering of simulations shown in this thesis were created using the OpenGL-based renderer.

5.3.3 Adaptability and Extendability

In `MuRoSimF` rendering can be extended and adapted easily.

An existing rendering system can be extended or customized by providing new `PropertyRenderers` or exchanging existing ones.

Besides this, the complete rendering system can be exchanged by another one. This work is greatly facilitated by the fact, that the basic interaction of the `RenderManager`, the `ObjectRenderer` and the `PropertyRenderer` does not make any assumptions on the way data is stored within the rendering system used. If the rendering system provides a storage of its own, as many scene graph APIs do (e. g. OpenSG[2]or OpenSceneGraph[3]), this can be used by

[2] http://opensg.vrsource.org/trac
[3] http://www.openscenegraph.org

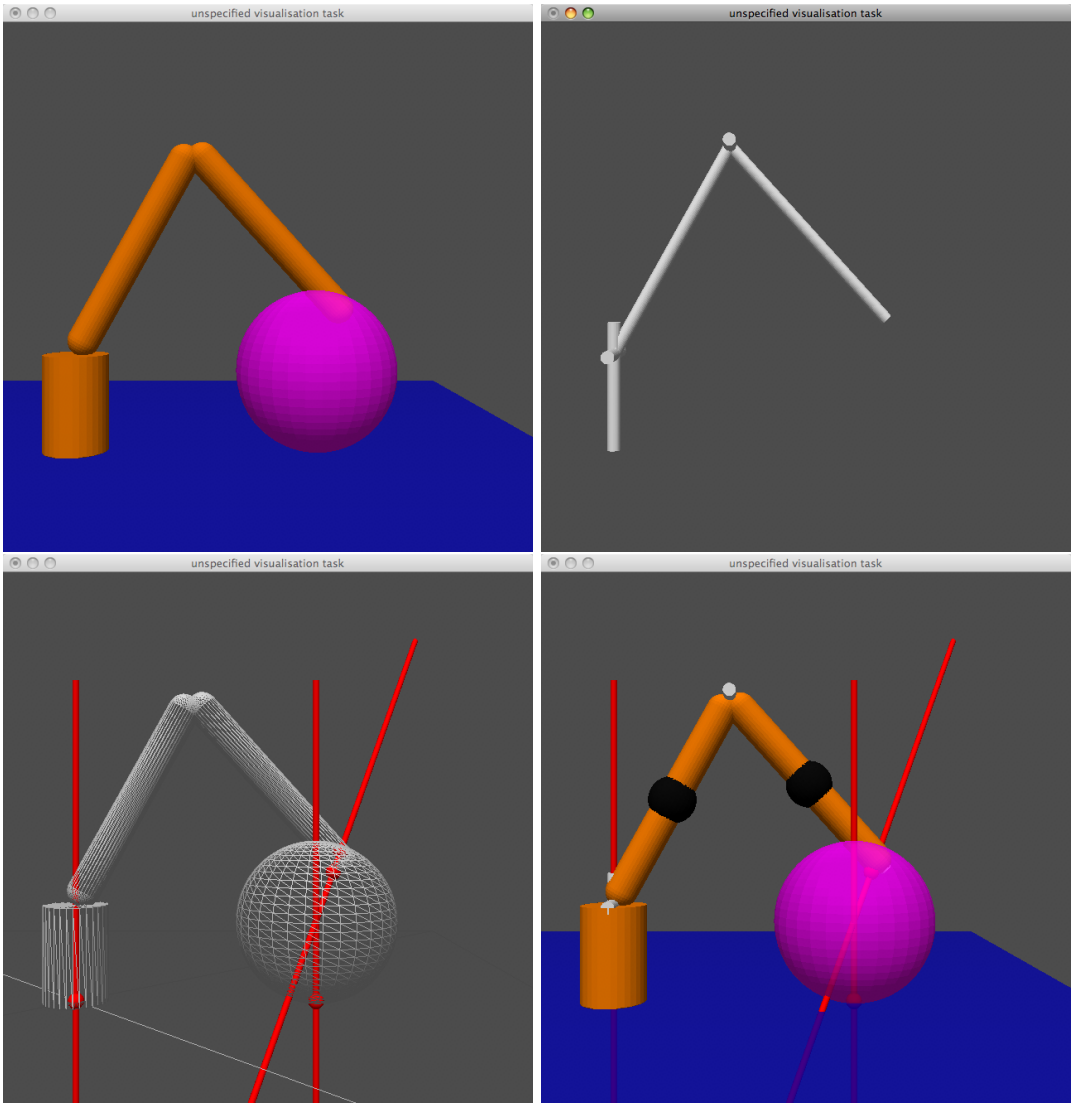


Figure 5.7: Examples for different property renderers. A scene of a simple manipulator colliding with an object is displayed in several different ways. Upper left: outward appearance of the scene. Upper right: Kinematical structure of the manipulator. Lower left: Collisions between manipulator and environment (including normals of the collisions). Lower right: Scene augmented with kinematical structure, collisions and center of mass for each link.

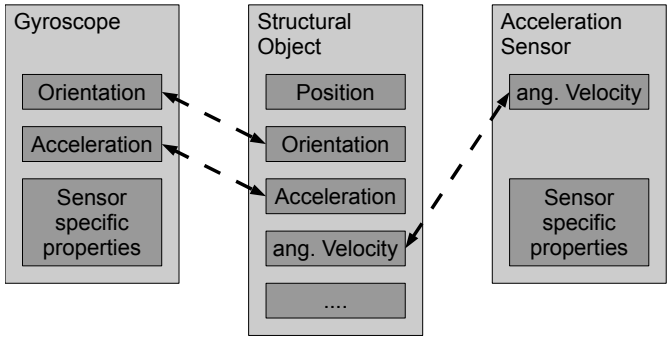


Figure 5.8: A gyroscope and an acceleration sensor attached to a structural object of a robot. The sensor objects only share the properties relevant to the sensor. Shared properties are marked with dashed arrows. If sensor measurements are to be augmented into the visualization of a simulated scene, it is necessary that the sensor shares the structural object's position and orientation even if they are not needed for the simulation of the sensor itself.

adopting the setup- and rendering methods in the derived classes appropriately. In case a scene graph is to be used as rendering system, the rendering methods of the `PropertyRenderers` will be used mainly to change values in the scene graph's nodes while the rendering itself is done by the scene-graph after the properties have been set.

5.4 Sensor Simulation

Sensor simulation is based on specialized sensor objects which can be attached to the robot's structure using the concept of explicit equality (cf. Section 4.1.2). After attaching the sensor-object to an object of the robot's structure, the sensor will share all properties of the object relevant to the sensor (Figure. 5.8). The sensor object may hold additional properties only relevant to the sensor. By using this concept, modeling of the robot's structure (which is mainly of interest to the collision detection and motion simulation) is decoupled from the sensor simulation, thus allowing further sensors to be added to a robot's structure without changing the structure itself. At the same time, other algorithms providing the properties measured by the sensor need not know about the added sensor as the property is shared transparently.

5.4.1 Simulation of Internal Sensors.

Internal sensors like gyroscopes, accelerometers, joint encoders or compasses are simulated by sharing specific values with the object the sensor is attached to. Depending on the kind of sensor, additional information about the sensor (e. g. the measuring axis of the sensor) or the environment may be needed to calculate the value measured by the sensor. Examples for the simulation of internal sensors are given in Table 5.9.

So far, the models of the sensors do not consider errors occurring during the sensing process like saturation of the sensor, non-linearity or noise. These are simulated in a two step process. In the first step, the sensor's *physical value* is transformed to the *output value*. Arbitrary functions describing the sensor's characteristic curve (e. g. box constraints or more complicated models)

Table 5.9: Calculation of the values of internal sensors.

Sensor	shared values	additional information	calculation of sensor value
Accelerometer	linear acceleration $\dot{\mathbf{v}}$ Orientation R	axis of sensor $\mathbf{a}$ vector of gravity $\mathbf{g}$	$s = (\dot{\mathbf{v}} - R^T \cdot \mathbf{g}) \cdot \mathbf{a}$
Gyroscope	angular velocity ω	axis of sensor $\mathbf{a}$	$s = \omega \cdot \mathbf{a}$
Joint position encoder	joint position q		$s = q$
Joint velocity encoder	joint velocity $\dot{q}$		$s = \dot{q}$
Magnetometer	Orientation R Position $\mathbf{r}$	axis of sensor $\mathbf{a}$ magnetic field[a] $\mathbf{B}(\mathbf{r})$	$s = \mathbf{B}(\mathbf{r}) \cdot (R \cdot \mathbf{a})$

[a] Note that the value of $\mathbf{B}$ is not provided by the simulation of the sensor, but must be provided by a different module of the simulation. By this the sensor-simulation itself is unaware of the way the magnetic field is simulated. The simplest possibility would be to provide a constant value. More complicated models, e. g. approximating the influence of motors, can be implemented alternatively.

can be applied for each sensor. Additionally noise may be added to the output value. To simulate effects of an analogue-digital-convert (ADC), the output value is further processed by limiting the value to the simulated ADC's range and finally transforming the value to an integer with a definable resolution, which is called the *digital value* of the sensor. All three values can be accessed by the robot control software, depending on the level of abstraction required for the simulation experiment.

5.4.2 Simulation of Cameras

Realtime simulation of cameras is based on the OpenGL visualization presented in Section 5.3. Cameras are described by two kinds of parameters: The camera's extrinsic parameters define where the camera is situated in space. When attaching a camera sensor to an object of the simulation model, the sensor will share the object's position and orientation, thus the camera is moved around, if the robot is moving. The camera's intrinsic parameters define, how the scene is seen by the camera. In the simplest case, these are the horizontal and vertical aperture angles of the camera.

By using this simple model, it is possible to simulate a pinhole camera providing perfect images. Images of this kind can be used for testing high level vision components working on images after image pre-processing.

More realistic simulation of a camera requires considering effects like distortion of the image caused by the lens or the image sensor. Distortions caused by the camera's lens can be described using more complicated models for intrinsic parameters. In the current implementation, the model used in the *Camera Calibration Toolbox for Matlab* [35] has been chosen. This model, which is based on [37], describes tangential and radial distortion caused by the lens by a 6th order polynomial and also covers decentering of the lens.

As a real image sensor does not take an image instantaneously, a certain amount of image blur can not be avoided when using a real camera. Depending on the shutter speed of the camera and the kind of sensor used in the camera, different approaches must be considered. Currently only general Gaussian blur is provided.

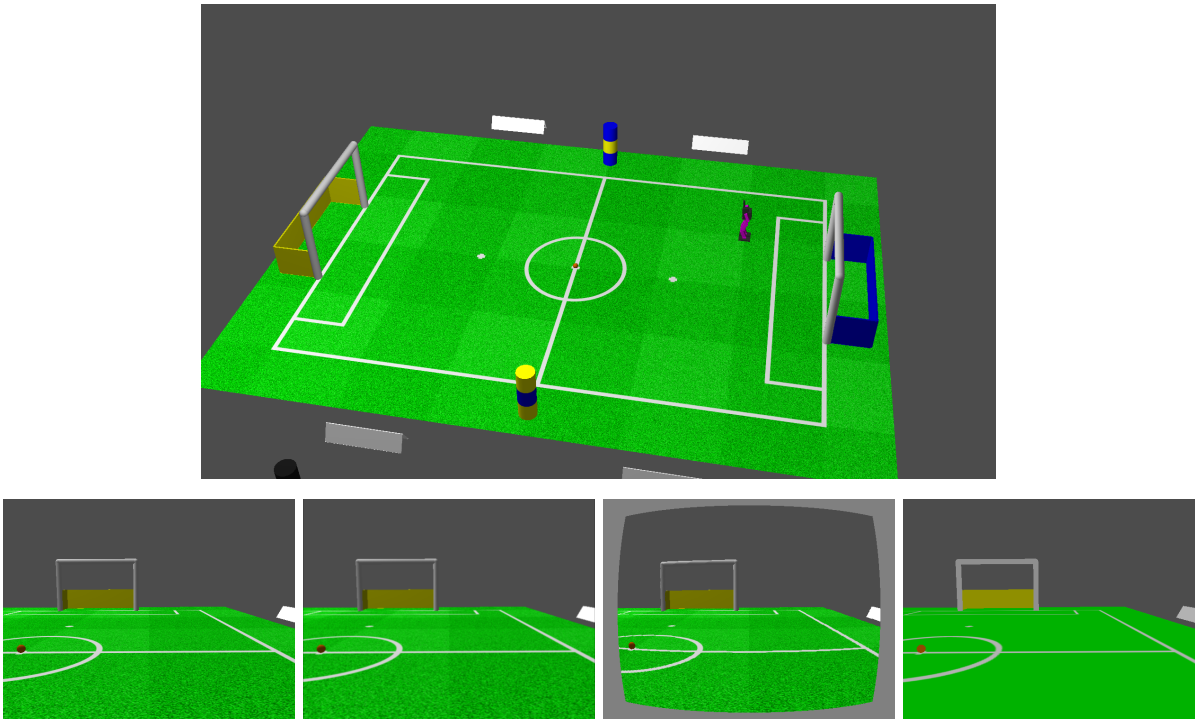


Figure 5.9: Examples for camera simulation. Top: a simulated scene. Bottom (from left to right): scene rendered from the simulated robot's point of view, additional Gausian blur, lens distortion, color coded image.

To provide more specific simulation of camera errors, advanced rendering techniques like the ones used to simulate motion blur and rolling shutter effects in [113] could be integrated. If these kinds of effects are to be considered by the simulation, additional information like the camera's velocity must be provided to the camera simulation, which can easily be achieved by the modeling techniques used in `MuRoSimF`.

Simplified Camera Simulation

Depending on the current use case, it may be of interest to simulate the camera in a simplified way.

If robots are situated in a strictly color coded world (as it is the case in the RoboCup soccer competitions), the vision subsystem often incorporates stages for color detection leading to clusters of equally colored pixels. In simulation experiments this preprocessing stage may be skipped if the simulated camera's images only consist of pixels of the environments standardized colors without any effects like shading, light or shadows. By adopting the rendering parameters of the camera simulation accordingly, such images can be generated.

A further simplification and higher abstraction of the simulation can be reached by no longer simulating the images created by the camera. Instead, coordinates of objects of interest can be generated directly. Three different levels of abstraction are provided: Simulation of the object's coordinates in the cameras image plane, in the robot's base coordinate frame or in world coordinates. The highest abstraction level will later on be called *oracle*. All three abstractions are no longer camera specific, but instead can be used as a general abstract sensor.

5.4.3 Laser Range Finders

Simulation of laser range finders is subdivided into two parts: Calculation of the rays, leading to a length and (in case of a hit) a material for each ray and simulation of errors. Both parts can be handled independently and (for perfect sensors) the second part may be skipped.

Simulation of the Rays

Simulation of a laser range finder requires calculation of the intersections of the range finder's rays with the objects present in the scene.

Simulation based on the depth buffer

One approach used in several simulations is using the depth buffer present in the graphics hardware for the calculation of the length of the rays. Utilizing the OpenGL based rendering system this approach has been implemented in the present simulation. After rendering the scene from the range-finder's point of view, the depth-buffer is read back using the OpenGl `glReadPixels` function. The information present in the depth buffer is not the length of the view ray for each of the pixels of the image, but the orthogonal distance z of the object from the viewing plane distorted by a perspective projection, depending on the viewing volume used for the rendering process[131]. The distance z can be calculated by

$$z = -\frac{f \cdot n}{\tilde{z} \cdot (f - n) - f} \tag{5.27}$$

with f and n being the distance of the far resp. near clipping plane from the viewing plane[4] and $\tilde{z}$ being the value read from the depth buffer (see Figure 5.10 for details). To obtain the length of the view ray, the ray's direction α must be taken into account, leading to

$$l = \frac{z}{\cos \alpha}. \tag{5.28}$$

The precision of the calculation of the ray-length is limited by the resolution of the depth-buffer. This resolution depends on the concrete OpenGL-implementation of the computer the simulation is ran on and may be out of control of the simulation.

The rays simulated by this method do not have an equal angular distribution (as it is the case in most laser range finders). Instead, adjacent pairs of rays have equal distance in the projection plane. Thus, it is necessary to add an interpolation stage after the calculation of the length of the rays.

As the rendering is based on the pin-hole model, the maximum opening angle must be well below 180°. If sensors with a wider opening angle are to be simulated, multiple runs of the algorithm for different angular ranges must be performed.

There is no direct way to determine the material of an object hit by a ray. Material information is provided by encoding the object's material in the color during the rendering process (instead

[4] The near and far clipping plane are planes parallel to the projection plane defining the part of space being projected. The viewing plane is the plane parallel to the projection plane containing the center of projection. For a further discussion of the projection process see [54] or [132].

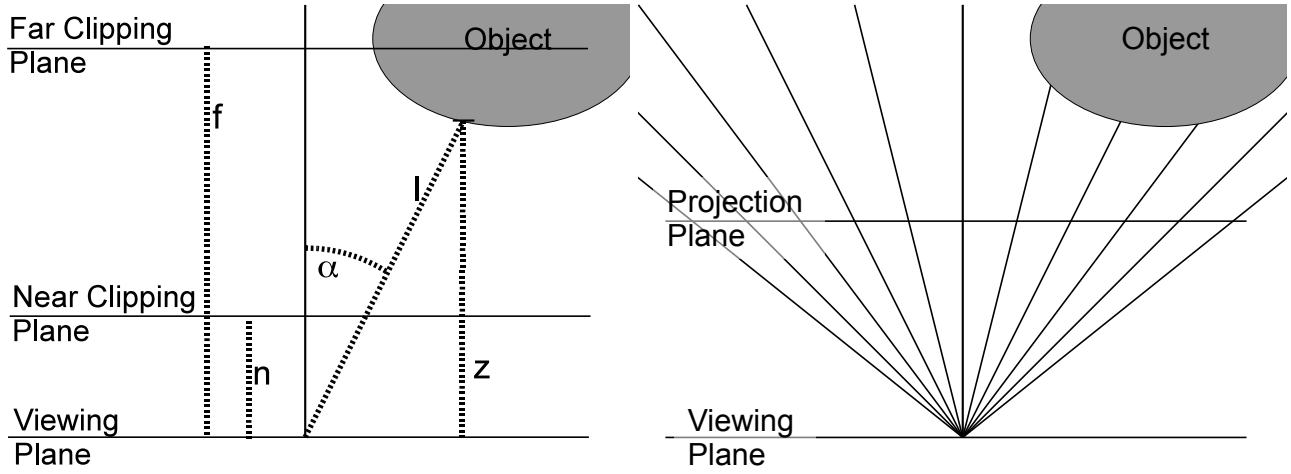


Figure 5.10: Simulation of laser range finder using the depth-buffer. Left: Calculation of length of ray. Right: Rays have equal distance in projection plane. Images adapted from [60].

of using the object's visible material). To determine the material a second call to `glReadPixels` is required to read the front buffer of the rendering.

Independently from the number of rays which are simulated, the method always requires that the whole scene is rendered.

Simulation based on ray object intersections

An alternative method for calculating the rays is the explicit calculation of ray-object intersections. To avoid intersecting each ray against each object of the scene the bounding-volumes provided by the collision detection subsystem of `MuRoSimF` are used.

This simulation method does not impose limits to the distribution of the sensor's rays. The material of an object hit by a ray is known directly, as the intersection calculation provides information on the object.

If more than one sensor is to be simulated, the method can be parallelized easily, as it does not write to any data in the simulated scene aside from the simulated sensor readings.

Comparison of methods

Two methods for the simulation of the length of the sensor's rays have been presented. Basically both provide the same information, namely the length of the ray and the object hit by the ray. Nevertheless, there are significant differences.

The most prominent difference is performance: as already shown in [60], the method based on the z-buffer has a better performance if many rays are to be simulated, while the ray intersection method performs better for sensors with few rays. Further measurements and discussions of the performance of the methods are given in Section 7.2.4. The actual break even is highly dependent on the machine the simulation is ran on and is dependent on the CPU and GPU of the respective computer. So from a point of view only considering performance, it is highly desirable to have both methods available to get optimal performance of a simulation on any machine.

Considering the accuracy more subtle differences can be found. The accuracy of the ray intersection method is only limited by the accuracy of the floating point format used. Arbitrary

distributions of the rays can be used without limitations. On the other hand, the accuracy of the z-buffer-method is limited by the resolution of the z-buffer. Further on, this method only allows for calculation of rays with an even, rectangular distribution.

Simulation of Errors

Errors can be simulated by using the same flexible error model as used for the scalar sensors presented in Section 5.4.1 for each ray. To allow for different materials, the model may be chosen for each ray individually, depending on the material of the object hit by the ray. Currently no advanced models, e. g. for crosstalk of neighboring rays are implemented, but can be added in an additional post processing step after the calculation of the rays.

5.5 Summary

In this chapter methods for the simulation of the motion and sensors of autonomous robots on different levels of abstraction have been presented. All methods have been implemented within `MuRoSimF` and can be combined into simulations, thus allowing the free configuration of simulations to provide simulation of different aspects of motion and sensing on different levels of abstraction.

For simulation of robot motion and for the collision detection components concepts were presented which allow an easy adaption of the respective methods for special needs. Motion simulation is based on a modular representation of the robot's MBS structure which can be extended to cover further structural elements without the need for changing existing models. By this it is possible to extend the currently present capabilities easily for additional needs, e. g. elastic elements of modern bionic robots (e. g. [88]) or wobbling masses of biomechanical systems (e. g. [137]). The components for collision detection and handling can be adapted easily to different needs, e. g. special object shapes or bounding volumes.

A unique feature is the capability of combining different methods for 3D and 2D kinematics and/or dynamics motion simulation within the same simulation. None of the simulations presented in Chapter 2.3 provide a comparable level of flexibility: Only Webots allows the combination of 2D and 3D simulation methods, but only the 2D methods may be exchanged by a defined interface. The only simulation with defined interfaces for exchange of algorithms is OpenHRP, but the interfaces are predefined for few aspects, namely 3D dynamics simulation, collision detection and visualization.

Several methods were presented for the simulation of internal and external sensors. Using the modeling techniques described in Chapter 4, sensor objects can be attached transparently to the robot models and provide access to all properties required by the respective methods for sensor simulation. Due to this decoupling the sensor simulation methods are completely independent of the methods used for the simulation of the robot motion. This enables a transparent exchange of either kind of method without affecting the other, thus allowing to combine methods with an adequate level of abstraction for a given task.

6 Integration and Execution of Models and Simulation Methods

In this chapter it is discussed how an executable simulation based on `MuRoSimF` is integrated using the modeling methodology and the simulation methods presented in the previous chapters. Further on it is discussed, how a simulation can be executed (possibly multi-threaded), how the simulation can be connected to the simulated robot's control software and how data created by the simulation can be further accessed and manipulated.

6.1 Definition of a Simulation

As discussed in Section 3.3.2 setup and execution of a simulation are structured into three phases: setting up the models, setting up the simulation methods and running the simulation.

In phase one the models describing the simulated robots and their environment are created. All models are registered with the simulation for later use.

In phase two the instances of simulation methods are created and connected with the models. Additional, method dependent information, may be added to the models during this phase. To run the simulation, it is necessary to execute these methods in the proper order and with the proper rate. This information is provided by the simulation's *schedule* which is set up during phase two. For the following discussion, each instance of a simulation method being executed in a running simulation will be called *task*.

The execution of each task is defined by a starting time (the first time the task is executed) and the task's specific period. To structure the schedule further, it is possible to aggregate tasks with the starting-time and period which are to be run in a distinct order in a so called *sequence*. Likewise, it is possible to structure tasks and sequences with the same starting time and period which can be executed in an arbitrary order and which have no mutual dependencies in a so called *parallel set*. Based on the definition of these parallel sets, it is possible to execute the simulation on multiple CPUs. Whether the tasks are parallelized though, depends on the runtime environment the simulation is executed in.

After setting up the models and the schedule, the definition of the simulation is finished and may be executed. Models, simulation tasks and schedule are aggregated into a single simulation object.

6.2 Executing a Simulation

Execution of the simulation is handled separately from the definition by so called *runners*. The runner is used to create a runtime environment for the execution of the simulation. If the simulation requires visualization, the runner will also create and manage the required windows and graphical contexts.

After setting up the visualization, the runner starts the execution of the simulation' s schedule. The simulation may be run synchronized to the computer's clock (in real time or at a defined relative speed). Execution of parallel sets is also handled by the runner. To provide some control on the CPU load of the simulation, the maximum number of threads can be limited. In order to avoid repeated creation and destruction of threads, a single thread pool is used providing a

predefined number of threads. If a parallel-set contains more tasks than there are threads, the tasks of this set are queued up and handled subsequently whenever one of the threads of pool becomes idle.

For interactive simulations the runner also handles user input through menus as well as through drag-and-drop interaction in graphical windows. To allow for interaction, the simulation must provide a set of so called `Interactors`. `Interactors` can be registered for compound objects or for single objects. Whenever an object is clicked on or dragged in an interactive window, the runner will check, if there is an `Interactor` registered for this specific object. If no object specific `Interactor` is present, it is further checked, if there is an `Interactor` for the compound object containing the selected object. In case an `Interactor` is found it will be notified of the interaction event and subsequently manipulate the object or compound object according to the user input.

6.3 Connecting to Robot Control Software

To connect a simulated robot to control software, specialized tasks called *controllers* are used. Depending on their purpose, controllers can be connected to different parts of the robot, e. g. the actuators or any sensor. A controller always can be connected to a communication device to provide communication with the external software. Different communication devices like serial ports or TCP connections are provided and can be exchanged transparently. An example setup is shown in Figure 6.1.

Communication with external control software always is asynchronous, the controller never blocks or waits for incoming messages. Thus this kind of connection only is suitable for high level control of the robot.

Low level control software, which is to be run with a fixed rate with respect to the simulation, must be executed as a simulation task and can be implemented in a controller. Besides programming a robot specific controller class, which is linked to the simulation, it is also possible to use a generic controller class, which is able to load the controller code from a shared library. For this purpose a simple procedural interface has been defined which must be implemented in the library. The interface consists of a set of functions for time stepping the controller, for setting current sensor values, for getting motion requests on servo-motor-level (desired position, velocity, acceleration or force, depending on the kind of controller) and for communication with the high level control software. To allow easy transfer of the controller software to a real robot, the same interface can be implemented on a micro controller. An example for this application is given in Section 7.1.5.

6.4 Data Interface

As the models of the simulated robots and their environment are based on a strict hierarchy (see Chapter 4), each property of any object of the simulation can be named uniquely by the triple (Compound Object - Object - Property). The simulation object provides a programming interface allowing to access each property by its unique name. This interface can be used by special simulation tasks, e. g. for logging of simulation data or for automatization.

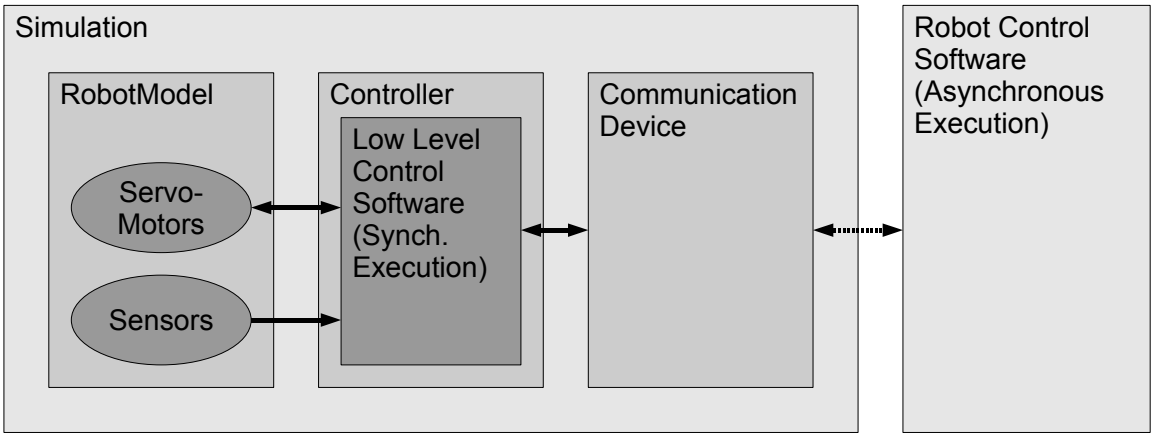


Figure 6.1: Integration of simulation and control software. The controller provide asynchronous communication to the external high level control software. Additionally it can execute low level control software synchronously.

6.5 Summary

In this chapter the mechanisms used to integrate and execute a simulation were presented. `MuRoSimF` provides a flexible way of defining the schedule for a simulation during the setup of the simulation to allow parallel execution of the simulation on multiple CPUs. Only when running the simulation, though, the number of threads to use is defined, thus allowing to limit the simulation to a maximum number of CPUs. Two distinct methods were presented for the asynchronous and synchronous integration of robot control software with the simulation.

7 Applications and Results

In this chapter several simulations are presented which have been created using the methodology, modeling concepts and simulation methods presented before in this thesis. Each of these application examples was created using MuRoSimF, the Multi-Robot-Simulation-Framework, which provides tools for modeling and simulation of robots on different levels of abstraction. A distinct feature of MuRoSimF which has been used in all application is the capability to choose and combine? simulation methods on different levels of abstraction within one simulation.

The first four examples are simulations of existing autonomous robots. All simulations has been used to support the development of control software for the respective robots. Each simulation makes use of the possibility to choose an adequate methods for motion and sensor simulation, sometimes even combining several such methods in one simulation simultaneously. Using the controller concept of MuRoSimF it has been possible to integrate the simulations transparently with the respective control software.

The humanoid robot simulation presented in Section 7.1 provides motion and sensor simulation for a team of humanoid robots on different levels of abstraction. A distinct feature of this simulation is the possibility of simulating several robots simultaneously using different levels of abstraction for the motion simulation. The motion simulation has been validated for each level of abstraction to allow a qualified decision whether simulation method to choose for a given application.

In Section 7.2 a simulation for search and rescue vehicles is presented. This simulation provides different methods for the simulation of the robot's laser scanners, which can be exchanged transparently.

A simulation for a heterogeneous team of robots with different modes of locomotion is discussed in Section 7.3. The main focus of this simulation is developing and testing of high level control software to coordinate heterogeneous teams. To provide a simulation adequate to this task, different simulation methods for each robot's motion (biped walking resp. differential drive) have been combined.

In Section 7.4.1 a simulation of a quadruped robot is presented. This simulation has been used during the development of the robot to guide the mechanical design of the robot in order to provide versatile viewing capabilities. Further on the simulation was used to develop a basic motion generation software. For both purposes a simplified dynamics simulation has been used.

The final example, presented in Section 7.4.2 differs from the other simulations, as its focus is not software development for autonomous robots. Instead, a kinematical simulation and collision detection of an industrial pipe bending robot is discussed. This simulation is not used in an interactive manner, but instead integrated into the manufacturers control software. Most parts of the simulation were developed using standard components provided by MuRoSimF. The integration of a domain specific collision detection was enabled by the highly adaptable collision detection concept.

Table 7.1: Computers used in experiments.

	Computer	
	A	B
Operating System	MacOS X 10.5.6	Windows XP pro SP2
CPU	Intel Core 2 Duo	Intel Centrino Duo
Clock	2.4 GHz	1.67 GHz
RAM	2 GByte	1 GByte
Graphics Hardware	Intel GMA X3100 (chipset)	Intel 945 GM express (chipset)

Computers used in experiments

The performance measurements described in Sections 7.1.6 and 7.2.4 were performed on computers with different hardware and operating systems. The main features are summarized in Table 7.1.

7.1 Simulation of Humanoid Robots

In this section a simulation for soccer playing autonomous humanoid robots is presented. Due to the complex structure, the many degrees of freedom and the complicated dynamics of motion, simulating a humanoid robot is a challenging task. Many 3D simulators discussed in Section 2.3 have been used to this purpose.

As discussed in Section 2.3, it is desirable to simulate motion and sensor features of autonomous robots on different levels of abstraction. None of the currently available robot simulators provides simulation methods for humanoid robots other than forward dynamics simulation of the robot. Unlike this, the simulation presented in this section provides simulation of the robot's motion on a purely kinematic as well as a simplified dynamic level in combination with the simulation of the robot's sensors (see Section 7.1.2). The motion simulation has been validated to some extend in Section 7.1.4. A forward dynamics simulation has been added recently (see Section 7.1.7). The simulation not just allows the exchange of the simulation method (something which can be done for the OpenHRP simulation, too), but additionally allows for simulating different humanoid robots simultaneously within the same simulation using different methods.

The simulation has been used as one of the main tools for the software development of the Darmstadt-Dribblers RoboCup team since 2006. The simulated robot is discussed in Section 7.1.1, some applications of the simulation are shown in Section 7.1.5.

7.1.1 The Humanoid Robot Bruno 2008

The simulated autonomous humanoid robot *Bruno 2008* is based on a kinematical design with 21 degrees of freedom (see Figure 7.1). The same structure has been used for several predecessors of Bruno 2008, which differ in the servo-motors used, the sensors and the computational capabilities. Even though this section focuses on the current model, it should be noted, that the former versions also were simulated using the same methodology.

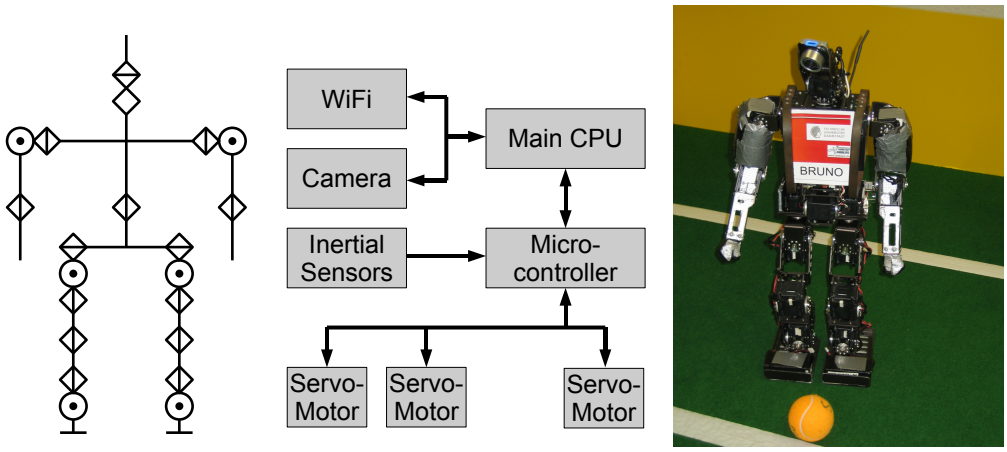


Figure 7.1: Humanoid robots used by the *Darmstadt Dribblers*: (a) Kinematical structure, (b) Distibuted computation, (c) Humanoid robot *Bruno*.

The robot uses Dynamixel RX-28 [122] and Dynamixel RX-64 [123] servo-motors. These actuators provide positioning based on an extended P-controller which can be adapted by several parameters (slope and compliance of the control law, maximum torque, velocity and voltage) allowing a wide variety of the behaviors of the actuator. All control parameters can be changed during runtime allowing adaption of the actuator's behavior, e. g. to switch it from stiff control to a more compliant mode to avoid damage if the robot is falling. Further on the actuators provide monitoring capabilities for the current position, velocity, temperature, torque and voltage. The actuators are equipped with an RS485 interface allowing to control up to 254 devices on one bus.

For stabilization and inertial navigation, the robot is equipped with 3 gyroscopes and 3 acceleration sensors. The main external sensor of the robots in the current configuration (used since 2008) is a pan tilt camera placed in the robot's head. Until 2007 the robots were equipped with an additional wide angle camera placed in the upper body [59].

Computing capacity is distributed to two computers coupled by a serial (RS232) connection[56]. Higher level (cognitive) functions like image processing, self localization, behavior control and team communication are executed on general purpose computers. In 2006 a pocket PC (running Windows CE 5.0) was used for this purpose. Since 2007 an embedded PC board is used (2007 running Windows CE, since 2008 running Linux as operating system). Lower level (reactive) functions like motion generation and stability control are executed on a 32-bit microcontroller.

The high-level control software of the robots is developed in C++ using the `RoboFrame` [116, 118] framework. It consists of a set of exchangeable modules for the key functions [57, 58]. These modules are executed in two separate threads for cognition (computer vision, self-localization and behavior control) and motion (communication with the microcontroller).

The firmware for the microcontroller is developed in C. It is running in hard real time generating setpoint trajectories for the robot's servo-motors every 10 ms [129].

7.1.2 Structure of the Simulation

The humanoid robot soccer simulation consists of a set of models and simulation algorithms connected to these models (see Figure 7.2). To model the environment, three distinct sets of objects are created: the ball, the static environment and interactive objects. All of these objects are registered with the collision detection, so that they may interact with other objects in the simulated scene, but only the ball is connected to a motion simulation algorithm as well, thus only the ball will react to collisions. Elements of the static environment are fixed and cannot be moved by the user, while user interaction with all other objects is possible.

An arbitrary number of robots may be added to the simulation. Each robot is connected to the collision detection to allow for interaction with the environment and other robots. For each robot a motion simulation task is added to the simulation. This task may be chosen arbitrarily for each robot, allowing the combination of different simulation methods for the robots. Currently the kinematic walking simulation and the simplified dynamics simulation are provided. To enable communication with the robot control software a robot controller is added for each robot. This task allows for integration with the high level control software as well as the firmware executed on the microcontroller (see Figure 7.3). The high level control software is conneted to the controller using a virtual RS232 line or a TCP connection, thus allowing asychronous execution like on the real robot. The firmware of the microcontroller is run synchronously within the robot controller using the mechanisms presented in Section 6.3. When using the virtual RS232 connection, the simulated controller is a completely transparent replacement for the real robot's controller.

Additionally for each robot the camera simulation and so called oracles providing ground truth data on the robot's and the ball's pose may be added. When using the camera simulation the framerate of the simulated camera as well as the kind of simulation (with or without lens distortion, with lighting or with plain colors) may be chosen for each camera arbitrarily. Simulated camera images and oracle data are transfered to the control application by a TCP connection. To feed the simulated camera information or the ground truth data to the control application, the control application must provide a specialized module to connect to the simulation. Thus the camera integration is not completely transparent, but the only difference from the application's point of view is the use of a different module providing the images. All other modules of the application remain unchanged.

The execution rates of the simulation tasks can be varied to some extend to adapt the computational cost of the application. The rates for ball motion simulation and *dynamical* robot motion simulation have to be the same as the rate of the collision detection. A step width of 1 ms has proven sufficient for a stable simulation. The robot controllers are run with an execution rate of 100 Hz, resembling the rate of the control function of the robot's firmware. Kinematic motion simulation is executed with the same rate. The execution rate of the camera simulation and the oracles can be chosen arbitrarily for each camera and oracle.

7.1.3 Simulation Model of the Humanoid Robot

The simulation model of the robot consists of a kinematical model with additional dynamical data and shapes. The basic kinematical structure of the robot and information on the shape, mass and center of mass of the robot's parts were derived from the CAD drawings of the robot.

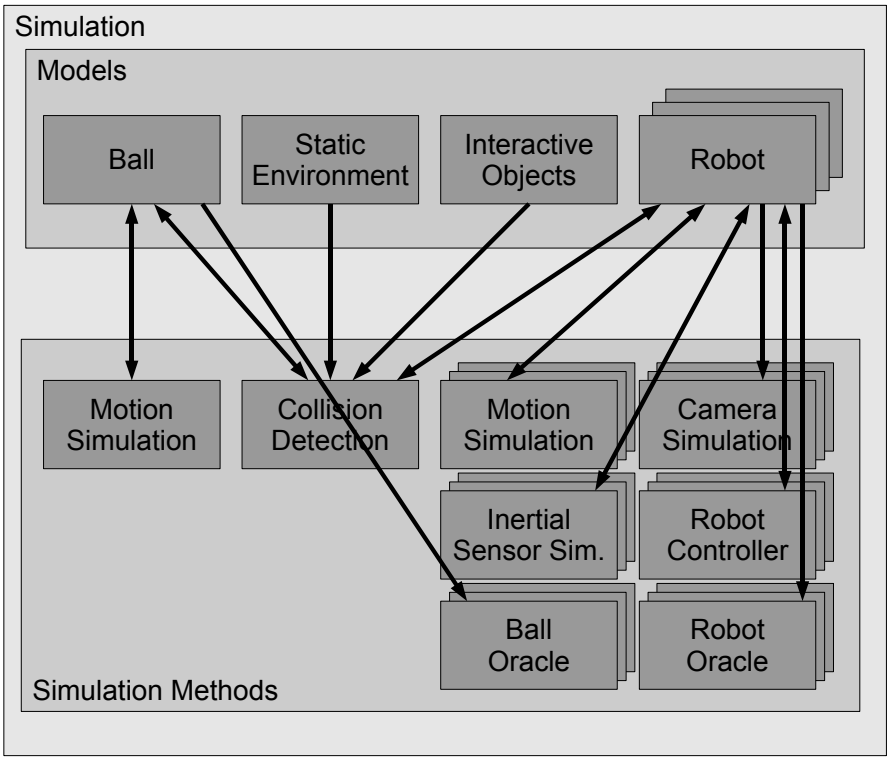


Figure 7.2: Structure of the humanoid robot soccer simulation: Arrows indicate dataflow between models and simulation algorithms. Note that for some objects there is no data flow from the collision detection back to the object, as these objects are static resp. only move due to user interaction. The so-called "oracles" are methods providing ground truth information on the pose of the robot resp. the ball. Methods associated with only one robot may be chosen arbitrarily for each robot.

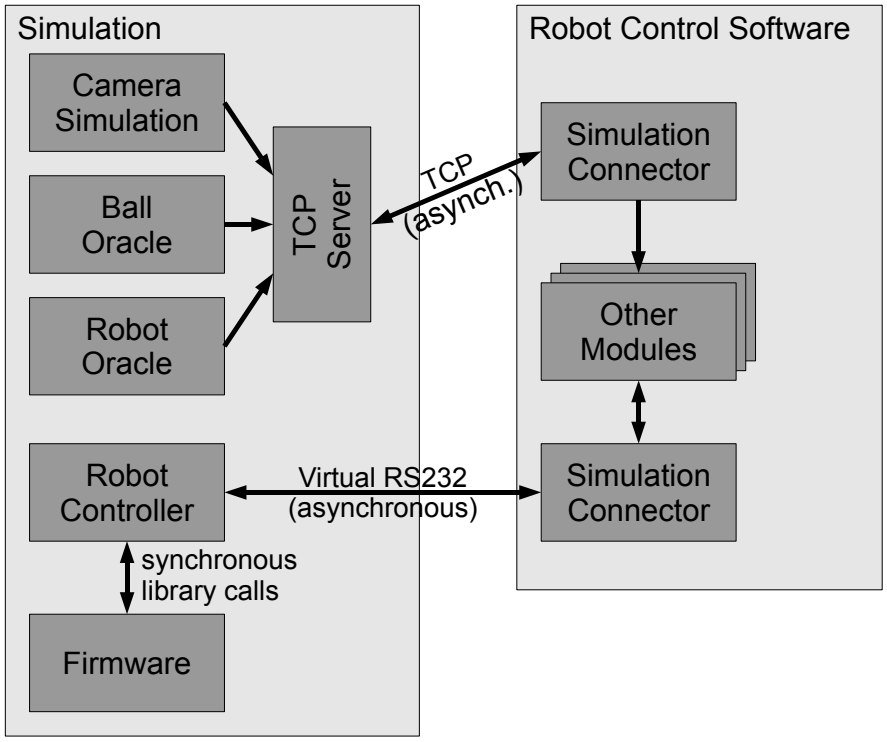


Figure 7.3: Integration of simulation with robot control software.

As no precise information of the inertia of the servo-motors and the other rigid bodies was available, the bodies were approximated as solid boxes for the calculation of the inertia tensors. The gear ratio and rotor inertia of the servo-motors were taken from the data sheets of the servo-motors[1].

During the validation of the motion simulation described in Section 7.1.4 the parameters describing the robot's contact model were fine tuned. Likewise the maximum torque as well as gear-friction of the servo-motors were estimated during the validation of the newly added dynamics simulation in Section 7.1.7.

7.1.4 Validation of the Simulation

In this section the different methods provided for motion simulation are validated. The main purpose of this validation is to gain information on the quality of the motion simulation in comparison with the real robot. This information can be used to evaluate the general fitness of the simulation for a given purpose. For a simulation which provides different simulation methods for the same task, in this case for the motion simulation, this evaluation is especially valuable, as it may be used a foundation for the selection of a specific method. Further on the validation process is used to improve the simulation model of the robot and to improve the simulation methods.

The validation follows the methodology presented in Section 3.2 of this thesis. It investigates two mayor aspects of the robot's motion, namely the distance covered by the real and simulated

[1] As the servo-motors use mechanically adapted motors, the data of the motor's base types which were provided by the motor's manufacturer had to be used.

robot, shaking of the robot's upper body while walking and the simulation of additional motions disturbing the operation of the robot.

Identification of Effects

Two major aspects of the robot's motion are relevant for the validation:

- Walking, as this is the robot's main mode of locomotion.
- Shaking of the robot's upper body, as this has a strong impact on the robot's sensing capabilities.

Inspection of the robot during operation reveals several effects influencing the main aspects of the robot's motion:

- Slipping of the robot's standing foot during walking: This effect has a strong impact on the distance covered by the robot during walking.
- Random turning of the standing foot: During the validation experiments it became obvious, that the robot also suffers from seemingly random turning motions during the contact phase.
- Shaking of the robot's standing foot during walking: The robot's standing foot does not always stand flat on the ground. This impacts the robot's sensor processing, as the relative orientation of the camera is calculated under the assumption of the foot staying put to the ground.
- Elasticity in the robot's joints: The servo-motors of the robot do not follow strictly the desired trajectory, thus causing the upper body of the robot to shake stronger during walking than expected.

Evaluation of Simulation Methods

Three simulation methods are provided for the humanoid robot's motion: kinematic walking simulation with and without slipping and the simplified dynamics simulation. All three methods are capable of simulating walking motions, but their capabilities for simulating effects influencing the motion are limited: The simplest method (kinematic simulation without slipping) does not consider any of the identified effects. Kinematic simulation with slipping provides a slipping model, but does not take into consideration any further effects of the contact situation of the standing foot. The simplified dynamics method provides a contact model including slipping and shaking of the robot's standing foot. As none of the simulation methods cover effects of the joints, quantitative comparasion of the walking motion is only feasible for the motion of the whole robot, but not on joint level. Further on none of the simulation methods covers the effects of the random turning of the standing foot.

As the kinematic simulation methods do not cover any effect which may cause shaking of the robot's upper body, this aspect of the motion cannot be simulated by these methods. The simplified dynamics method, on the other hand, allows for shaking of the standing foot, so that a shaking motion of the upper body occurs in the simulation. As other effects like the elasticity

Table 7.2: Evaluation of simulation methods. This table summarizes, which aspects of the robot's motion and which of the effects influencing these aspects are expected to be simulated quantitatively (quant.), qualitatively (qual.) or not at all (/) by the provided simulation methods.

Simulation Method	Effects				Aspects	
	Slipping of standing foot	Random turning of standing foot	Shaking of standing foot	Elasticity in joints	Walking	Shaking of upper body
Kin. walking	/	/	/	/	quant.	/
Kin. walking w. slipping	quant.	/	/	/	quant.	/
Simplified dyn.	quant.	/	qual.	/	quant.	qual.

in the joints, which also has a strong influence on the upper body, are not covered by this simulation methoda quantitative comparison of the shaking of the real and simulated robot is not feasible.

The evaluation of the present simulation methods is summarized in Table 7.2.

Selection of Validation Methods

As discussed in the previous section, all three simulation methods allow for a quantitative comparison of the whole robot's motion. None of the methods covers the effects of the random turning, so it is only feasible to analyze the distance covered by the robot, but not the lateral motion of the robot. Thus, the validation method for the walking simulation will be a comparison of the distance covered by the real robot and the simulated robot for each of the methods.

The validation experiments will also be used to minimize the difference between the distance covered by the real and the simulated robot. For the kinematic simulation only the slipping parameter may be optimized, for the simplified dynamics simulation the experiments will be used to improve the contact model.

As no quantitative validation for the shaking motion of the upper body is feasible, only the quality of the motion will be assessed.

Validation Experiments

Distances covered by real and simulated robot

The distance covered by the real robot walking 20 steps in a straight line was measured ten times for 6 different walking speeds. The setup for the experiment is shown in Figure 7.4. During these experiments a random turning of the robot's standing foot was observed: During motion the robot sometimes, but with no noticeable pattern, turns the standing foot a slight bit, leading to a deviation of the walking direction. In Figure 7.5 the positions reached by the robot during the experiments are depicted.

The same experiments were repeated for the simulated robot with each of the three methods. For the kinematic simulation with slipping and the simplified dynamics simulation the experi-

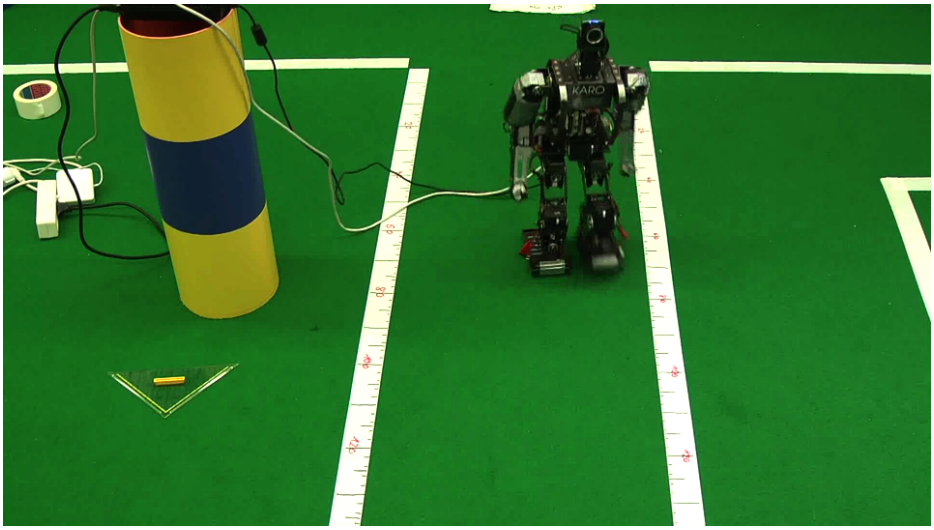


Figure 7.4: Setup of the validation experiment.

Table 7.3: Distances covered in experiments.

Method	Walking speed [mm/s]	50	100	150	200	250	300
Real Robot	avg. distance [mm]	234.4	523.8	736.4	988.5	1180.4	1294.3
	standard deviation	10.4	10.6	18.3	13.9	16.2	13.1
kinematic simulation	avg. distance [mm]	233	483.7	685.6	893.7	1092.3	1253.7
	standard deviation	0	1.16	0.52	0.95	0.95	1.16
kin. sim. w. slipping	avg. distance [mm]	246.9	512	734.8	963.5	1178.3	1354.3
	standard deviation	0.3	0	1	3.1	0.9	1.8
simplified dynamics	avg. distance [mm]	253.5	514.2	723.6	897.5	1080.6	1240.8
	standard deviation	2.2	4.1	5	2.8	3.5	4.7

ments were repeated while hand-tuning the parameters of the slipping resp. contact-model. The measured distances covered by the real and simulated robots are summarized in Table 7.3. To allow a better comparison of the covered distances, the distances covered by the real robot and the the robot simulated with the three methods are visualized in Figure 7.6 to 7.9.

Adaption of simulation

During the validation experiments limitations of the contact- and slipping models became obvious. To improve the results of validation, the following adaptions were made to the simulation methods:

- Kinematic simulation with slipping: To better reflect the slipping behavior of the robot, a minimum velocity was introduced to the slipping model. Only if the relative velocity of the foot is above this velocity, slipping does occur.

- Contact model for simplified dynamics simulation: The linear viscous friction model presented in Section 5.2.3 was extended to allow simple non-linearities by using two different friction coefficients below and above an adjustable border.

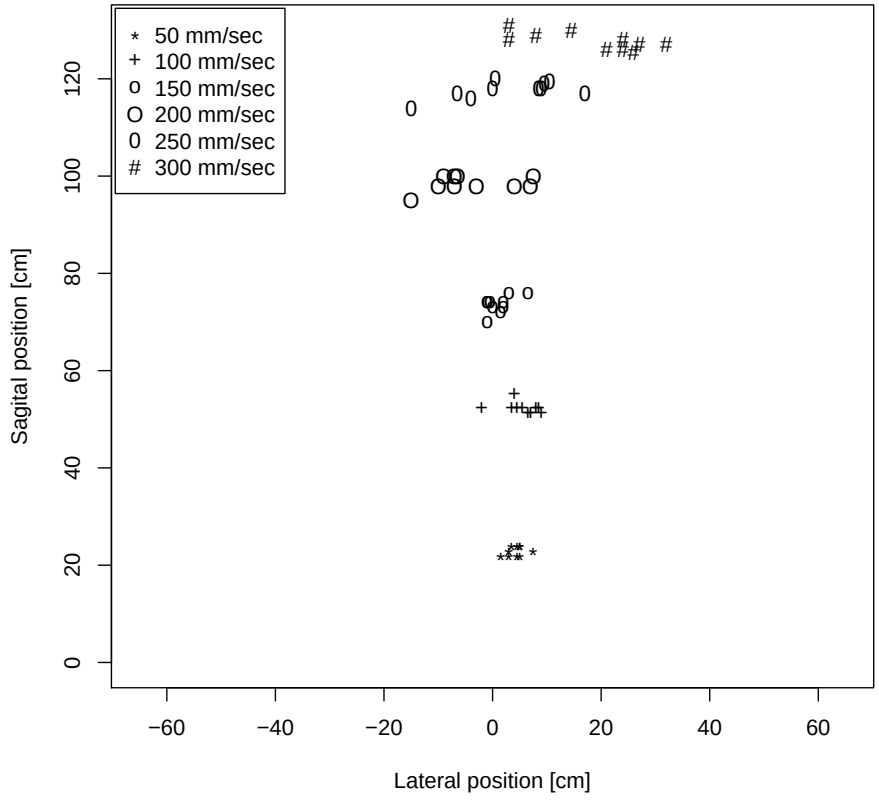


Figure 7.5: Positions reached by the robot during the validation experiments.

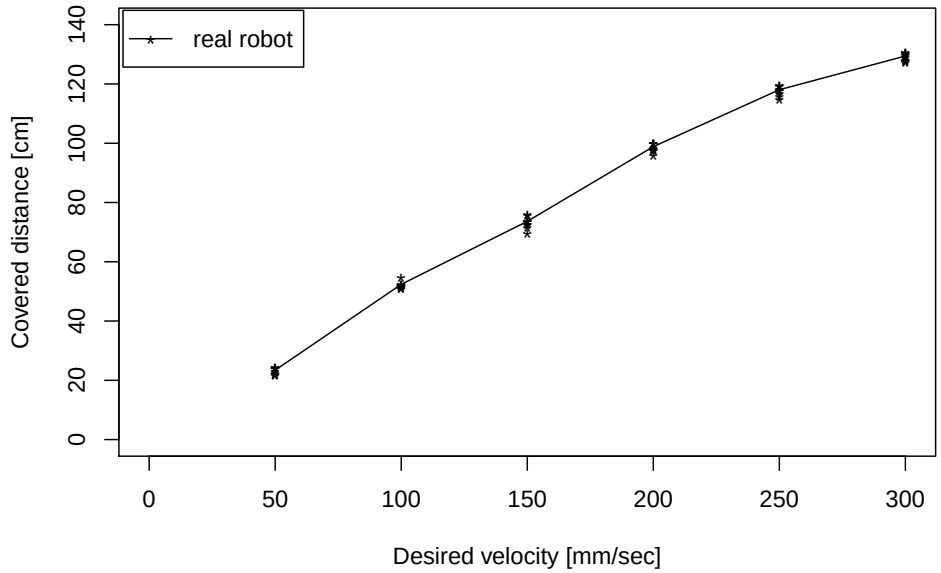


Figure 7.6: Distances covered by the real robot during the experiments. The average distances are connected by a line. Note, that this line is only drawn for better visibility of the average values. No measurements were made for other but the six velocities indicated.

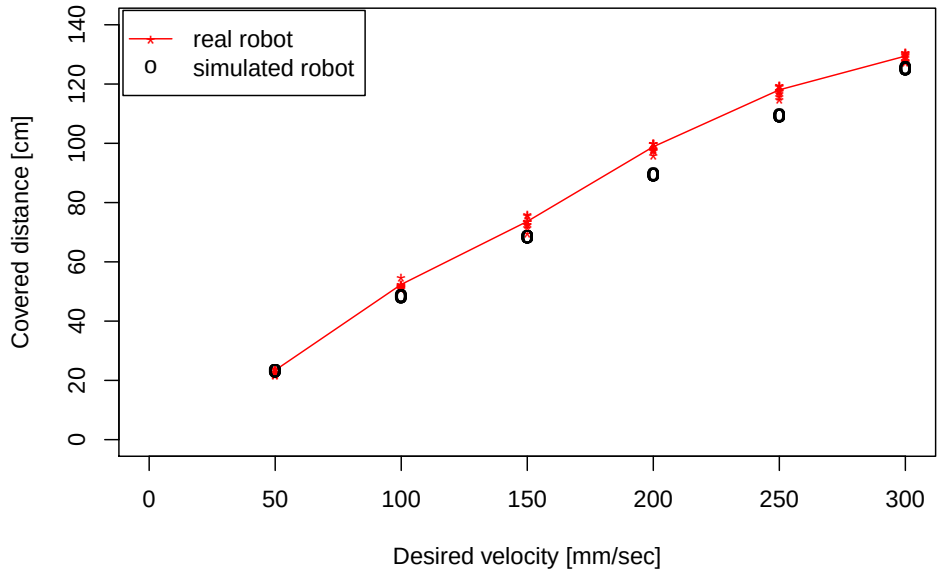


Figure 7.7: Distances covered by the robot simulated using the kinematic walking method without slipping. The distances covered by the real robot are shown as small red stars for comparison.

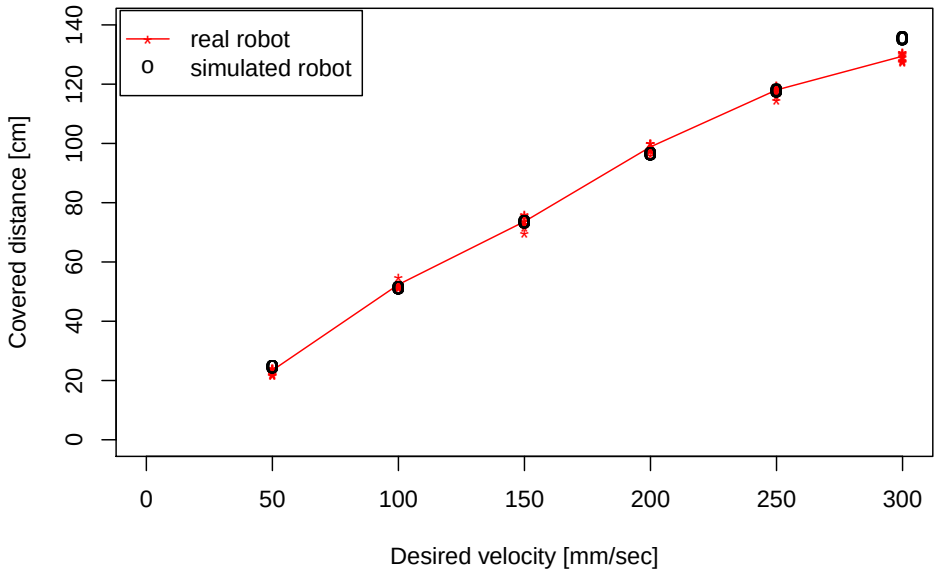


Figure 7.8: Distances covered by the robot simulated using the kinematic walking method with slipping. The distances covered by the real robot are shown as small red stars for comparison.

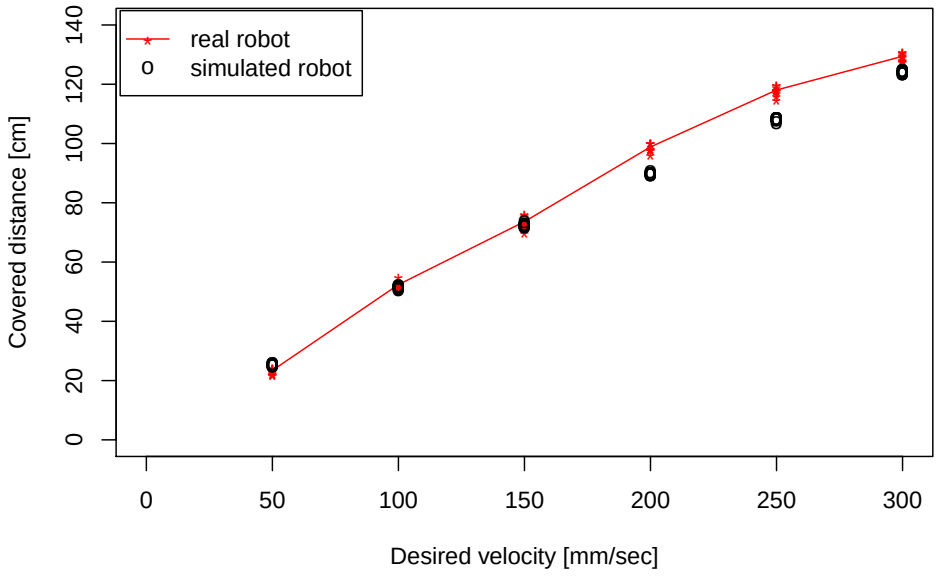


Figure 7.9: Distances covered by the robot simulated using the simplified dynamics method. The distances covered by the real robot are shown as small red stars for comparison.

Table 7.4: Simulation errors.

Method	Walking speed [mm/s]	50	100	150	200	250	300	all
kinematic simulation	mean error [%]	0.6	7.6	6.9	9.6	7.5	3.1	5.9
	maximum error [%]	0.6	8.0	7.0	9.7	7.5	3.2	9.7
kin. sim. w. slipping	mean error [%]	5.4	2.2	0.2	2.4	0.2	4.6	2.5
	maximum error [%]	5.4	2.2	0.3	2.5	0.2	4.9	5.4
simplified dynamics	mean error [%]	8.2	1.8	1.7	9.2	8.5	4.1	5.6
	maximum error [%]	9.7	3.6	2.5	9.7	9.3	4.7	9.7

Shaking motion of upper body

Inspection of the motion of the robot's upper body in the simplified dynamics simulation reveals, that there is a visible shaking of the upper body which is not caused by the mere motion of the robot's joints.

Discussion

To compare the performance of the three simulation methods, the relative error of the simulation compared to the average distance covered by the real robot has been assessed (see Table 7.4). This comparison leads to the following results:

- All methods produce an error when comparing the distances covered. This error is dependent on the walking speed for all methods.
- The maximum error of the pure kinematic walking simulation and of the simplified dynamics simulation is equally large (9.7%), the average error is comparable (5.9% resp. 5.6%). The kinematic simulation with slipping has a smaller error (maximum 5.4%, mean 2.5%).
- Both kinematic simulation methods have nearly no variance of the distance for repeated experiments. The simplified dynamics method has some variance, but it is still considerably smaller than the variance for experiments with the real robot.

Selection of a simulation method

None of the simulation methods provide a highly accurate simulation of the robot's motion. Nevertheless all of them are accurate enough for the applications with low or medium requirements on the simulation accuracy given in Table 3.1 like testing of behavior control or basic motion generation. Each method has specific advantages and drawbacks concerning the simulation of disturbances, so a method can be chosen according to these (cf. Table 7.5):

- If simulation of additional disturbances is not required by (or even harmful for) a specific simulation experiment, the pure kinematic simulation method can be used. A typical application would be testing the robot's behavior without the impact of additional errors caused by the robot's motion.
- If simulation of disturbances is required, kinematic motion with slipping or the simplified dynamics method can be chosen, depending on which disturbances need to be simulated.

Table 7.5: Comparison of simulation methods.

Method	Error in covered distance		Simulation of disturbances	
	mean	max	slipping	shaking
kin. sim	5.9	9.7	no	no
kin. sim. w. slipping	2.5	5.4	yes	no
simp. dyn.	5.6	9.7	yes	yes

The methods differ in the kind of disturbances they produce. As only the simplified dynamics simulation provides a shaking of the standing foot, only for this method the orientation of the real camera will differ from what the control software is able calculate from the robot's joint encoders. If this kind of effect is to be considered in the simulated experiment, the simplified dynamics method should be used. If only general slipping of the robot is of interest, either method may be used.

Suggestions for further improvement of the simulation

None of the simulation methods currently cover effects of the robot's joints and of the seemingly random turning of the robot. An additional issue is the error made by the simulation concerning the distances covered by the robot.

The random turning of the robot can be incorporated into both simulation methods by introducing random models for the contact situation. Establishing such methods will require measuring the turning for different motions and velocities, so that a statistical model of these motions can be created.

Effects concerning the joints like elasticity can not be covered by the methods used so far, as these methods all are based on a purely kinematic simulation of the robot's inner motion. The upcoming full MBS simulation of the robot will cover these effects (see Section 7.1.7), but has not yet been fully integrated into the simulation.

The errors made during the walking simulation can be reduced by introducing refined models for the slipping of the robot. For the simplified dynamics method, this can be done by altering the contact model to consider more different velocity ranges, for the kinematic simulation similar changes can be made to the slipping model.

Validation of camera simulation

Up to now only the simulation methods and models affecting the motion simulation have been validated. A formal validation of the camera simulation is still missing. As the quality of the images produced by the camera used on the robot is quite high and the camera has nearly no distortion, the major aspect of the camera, namely producing images of the surrounding scene still can be simulated (see Figure 7.10). It should be noted though, that test of parts of the software which handle special effects of the camera (like color segmentation) need to be adapted for the real robot, e. g. by providing a specific color table.

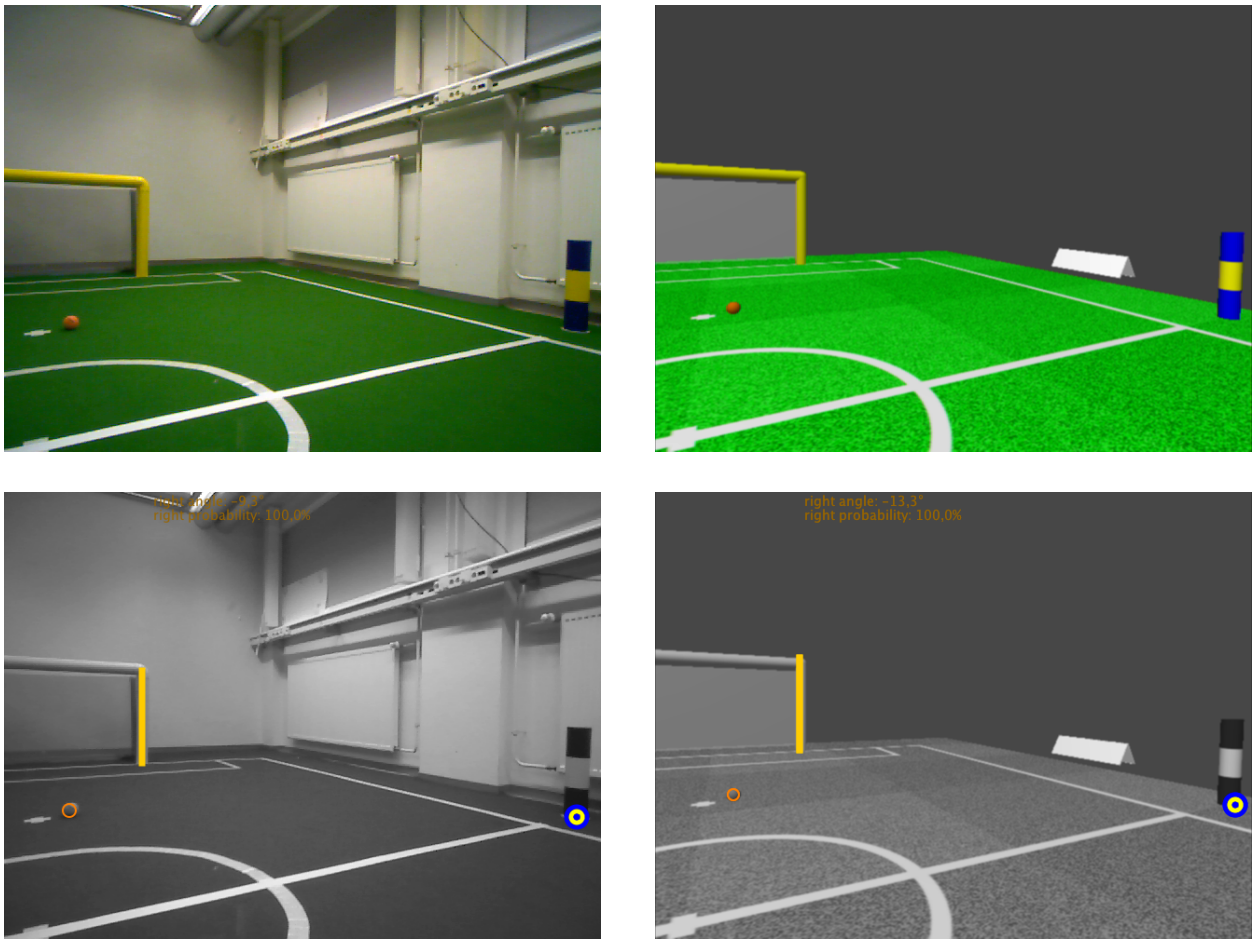


Figure 7.10: Comparison of real and simulated camera images. Upper row: real and simulated camera image. Lower row: camera images augmented with percepts of ball, goalpost and pole (for better visibility of the percept markers, the original image is shown in greyscale, nevertheless the image processing was done on the respective colored image). Even though the images produced by the simulated camera differ from those of the real camera, the images lead to comparable results on a percept level.

7.1.5 Applications of the Simulation

The simulation has been used successfully for testing different modules of the robot's control software.

Testing of Self-Localization

The simulation allows interactive testing of the robot's self-localization. For this purpose the robot's motion is simulated using either the kinematic or the simplified dynamics method (the later for additional undesired shaking motion of the camera). While monitoring the robot's control software, interaction with the simulation can be done easily by moving around the robot using the mouse. Using the simulation also the general usefulness of different sensor configurations (e. g. varying aperture angles of a camera or additional sensors like a compass) can be evaluated without modifications to the real hardware.

Testing of Behavior Control

The simulation has been used extensively in the development of the robot's behavior control for soccer games and technical challenges [61, 62, 139]. For these tests the kinematic motion simulation in combination with either camera simulation or oracles has been used. Example scenarios are given in Figure 7.11 and 7.12.

Development of Motion Generation

The simulation has been used during the development of a new motion generation system for the humanoid robots [129]. Extensive use has been made of the capability to run the real-time part of the robot's firmware within the simulation. This has enabled comfortable debugging of the firmware (including freezing the software any time during execution, a task which is nearly impossible on the real robot without the danger of damaging the hardware).

Further on the simulation provides capabilities for monitoring a robot's motion, thus allowing evaluation of motion trajectories before testing them on a real robot (see Figure 7.13).

7.1.6 Performance

To evaluate the performance of the humanoid robot simulation, the number of maximum number of robot and the maximum framerate of the camera simulation which could be simulated at real time were measured. Two computers with different properties were used for these experiments (see Table 7.1 for details).

Realtime performance of motion simulation

The realtime performance of the motion simulation was measured for the kinematical and the simplified dynamics methods either running with one or two threds (see Table 7.6). No camera simulation or communication with external software was activated during the measurements.

Two conclusions can be drawn from these measurements:

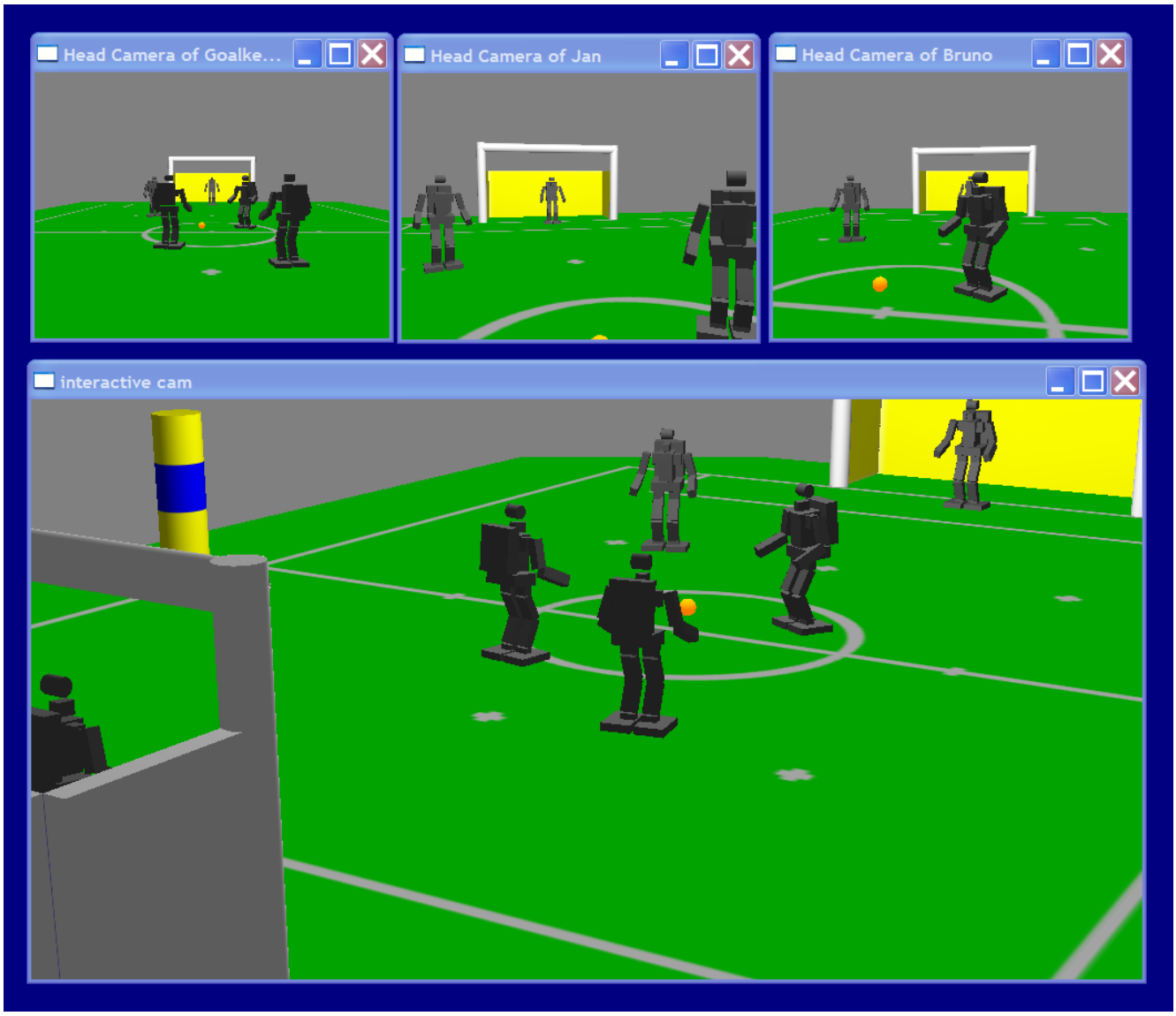


Figure 7.11: Testing of humanoid soccer behavior [62].

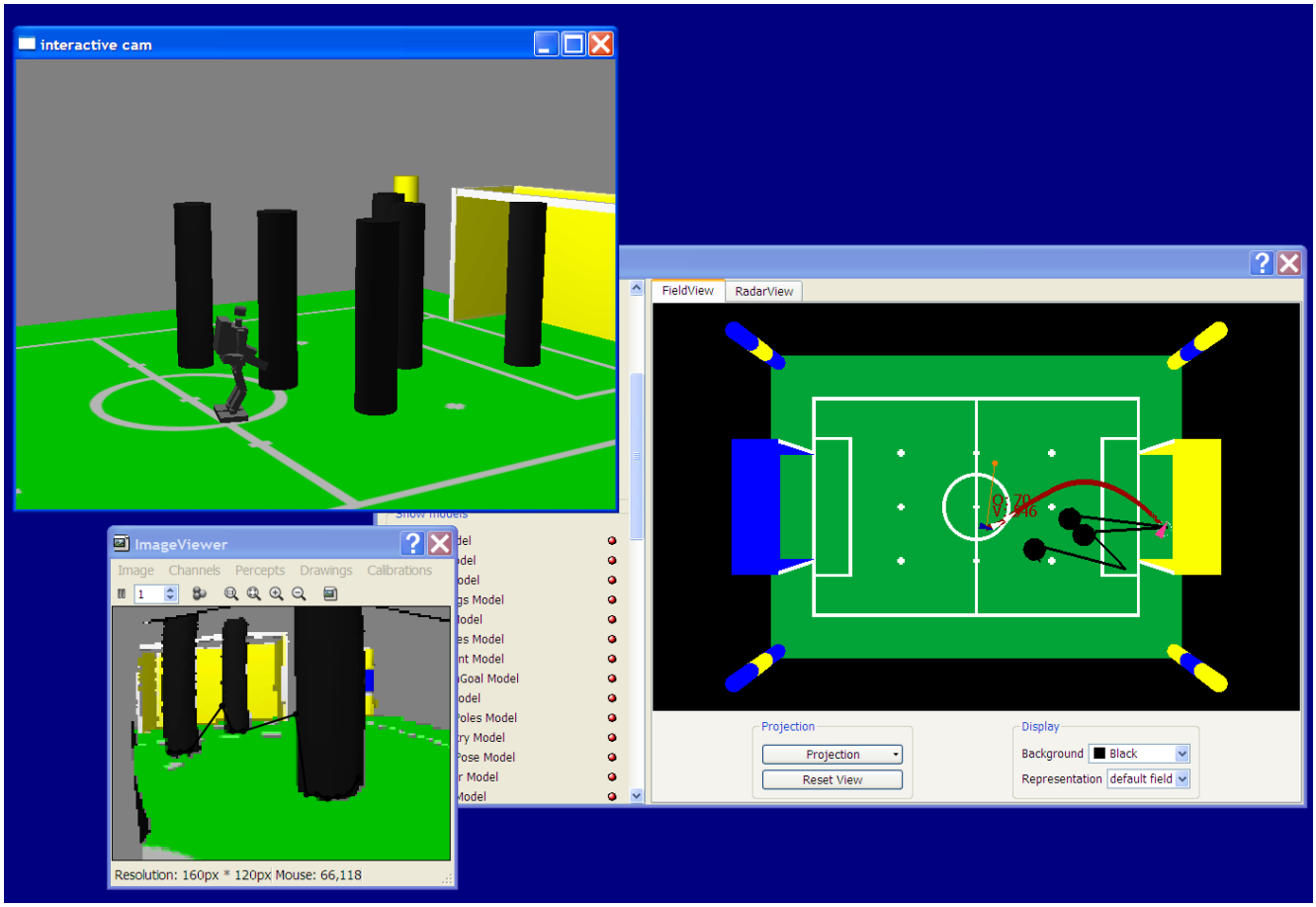


Figure 7.12: Testing of the dribbling challenge. Upper left: Simulated scene; robot and obstacles can be moved interactively by the user. Lower left: Simulated camera image of the robot. Right: World model from the robot control software. Black circles depict the obstacles detected by the robot, the red curved arrow is the robot's currently planned collision free trajectory [62].

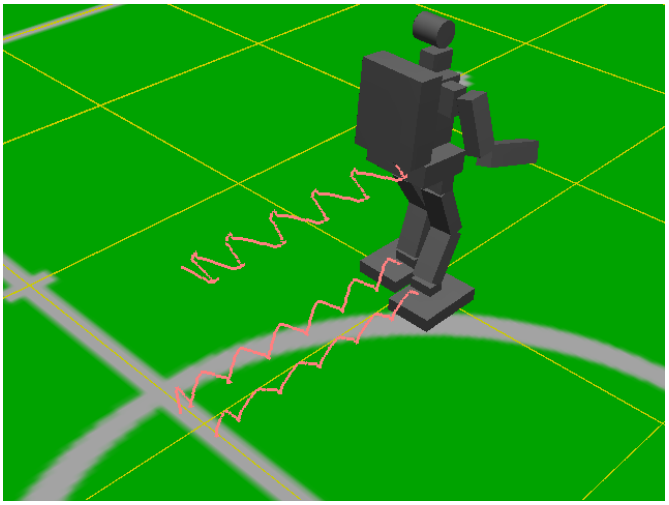


Figure 7.13: Visualisation of a robot's motion trajectories: The motion of the robot's hip and the feet is augmented to a kinematic simulation, allowing evaluation of the motion generation before testing on the real robot [117].

Table 7.6: Realtime performance (maximum number of simultaneously simulated robots) of the humanoid robot motion simulation.

Method	Computer (Threads)			
	A(1)	A(2)	B(1)	B(2)
Kinematic	23	25	14	14
Simplified Dynamic	10	12	4	6

- The kinematical walking simulation allows for a higher number of robots to be simulated, thus enabling a tradeoff between performance and accuracy of the simulation.
- Neither methods scales well with the number of threads. Two reasons were found for this:
  1. The collision detection, which currently is not multithreaded, dominates the CPU usage of the simulation.
  2. The calculation times of single executions of the motion simulation are very small, but have to executed very often. For this reason, the gain made by using threads is nearly compensated by the overhead of threading (e. g. synchronization).

Realtime performance of camera simulation

To measure the realtime performance of the camera simulation, a scenario with six humanoid robots (each with kinematic motion simulation) was used. No external communication was active during the measurements.

For the measurements the camera of one robot was activated and the maximum number of frames simulated at realtime was measured for several resolutions with and without distortion (see Table 7.7). The experiments show, that the maximum number of frames which can be simulated highly depend on the size of the simulated camera images. An even bigger impact though, has the CPU bound calculation of the camera's distortion, which currently is done after reading back the image from the graphics hardware within the rendering callback. Two solutions for this bottleneck will be investigated in the future: calculating the distortion during rendering (e. g. using texture mapping techniques as described in [127]) and calculating the distortion in a second thread.

Another effect, especially affecting the performance on computer B, was a significant delay when reading back images from the renderer (see Table 7.8). Note that the times given in the table do *not* include the time needed for rendering the scene.

Discussion

The performance results show, that it is possible to simulate even large teams of humanoid robots in realtime using an appropriate level of abstraction for the motion simulation. Camera simulation though, may become a bottleneck if large resolutions or many cameras need to be simulated.

For many application, the use of abstract sensors, as provided by the "oracles" for ball and robot pose, is an alternative to using a camera simulation. When measuring the performance of the motion simulation, additional oracles providing information for each robot at 100 Hz had no impact on the performance on either system.

Table 7.7: Real time performance (maximum frames per second) of camera simulation.

Method	Resolution	Computer A	Computer B
plain	160 x 120	100[a]	62
	320 x 240	86	30
	640 x 480	48	9
distortion	160 x 120	78	31
	320 x 240	36	11
	640 x 480	12	2

[a] 100 frames per second is not a limit of computer A. The simulation runner currently used allows not more than 100 fps for *one* camera. When simulating two cameras with a resolution of 160 x 120 in realtime, each camera has a framerate of 54 fps.

Table 7.8: Average time for image capturing [ms].

Resolution	Computer A	Computer B
160 x 120	2.6	3.1
320 x 240	3.9	11.9
640 x 480	7.8	48.3

7.1.7 Full Multi Body Dynamics Simulation

Recently a full MBS simulation of the humanoid robot has been developed, but has not yet fully been integrated into the soccer simulation [119].

Realization

The simulation is based on the forward dynamics simulation as described in Section 5.1.1. To provide the motor torques, a simulation of the servo-motors was developed. This simulation is based on the rough description of the control strategy of the motors in use that is given in the manufacturer's documentation [122, 123].

Validation

The simulation has been tested and validated for the simulation of the forward dynamics of the robot for the special case of a fixed base.

Identification of relevant effects

The main aspect of interest for the validation of the forward dynamics simulation are the joint angles of the robot's servo-motors during motion. These angles are affected by several parameters. Initially the following parameters were investigated:

- Maximum torque of the servo-motors.

- Friction of the gears of the motors.

For an ideal robot, these parameters should be equal for all servo-motors of the same type.

During the experiments it became obvious, that a further effect needs to be covered by the simulation, namely the inertia of the servo-motors' rotors.

Validation experiments

Two series of experiments were performed for the validation. In the first series of experiments each single servo-motor of the real robot's legs was moved along a predefined trajectory with each other servo-motor staying in its zero position. In the second series of experiments the real robot performed walking motions (with a fixed base and without contact) with several velocities.

During all experiments the desired positions and the real positions of all servo-motors were logged for later comparison.

The same experiments were repeated with the dynamics simulation. To do this, the measured desired positions were used as input to the servo-motor simulation. The positions of the simulated servo-motors were compared with those of the real ones.

During these experiments the parameters of the servo-motor simulation were adapted to improve the fitting of the trajectories of the real and simulated robot. Initially two parameters were considered: the viscous friction of the joints and the maximum torque of the motor. Both parameters were investigated for both types of servo-motor used in the robot, but it was not differentiated further by providing different values for each individual motor. During these experiments it became obvious, that the servo-motors driving parts of the robot with an exceptionally low inertia like the head or the lower arms, were accelerating much to fast. Two possibilities were considered to overcome this problem. The first one, namely choosing highly different parameters for the affected servo-motors was rejected, as this would not reflect the fact, that the motors in the robot are of only two different types. Instead the dynamics model of the robot was extended to cover the additional effect of the inertia of the motor's rotor, introducing the rotor inertia as another modeling parameter.

The validation of the simulation has led to a model which fits the real robot quite well (see Figure 7.14).

Discussion

Validation has led to a simulation model which fits the overall motion of the real robot's joints.

Further improvements can be reached by considering slight deviations (as are present in the real motors, too) of the single motors from the modeling values that were chosen equally for all motors of the same type. Another effect which has not yet been covered in the model is the additionally elasticity of the gears used in the servo-motors.

7.1.8 Summary

In this section a simulation for a team humanoid robots has been presented. The simulation has been used successfully during the development of software modules for behavior control, localization and motion generation of humanoid robots. A distinct feature of this simulation is

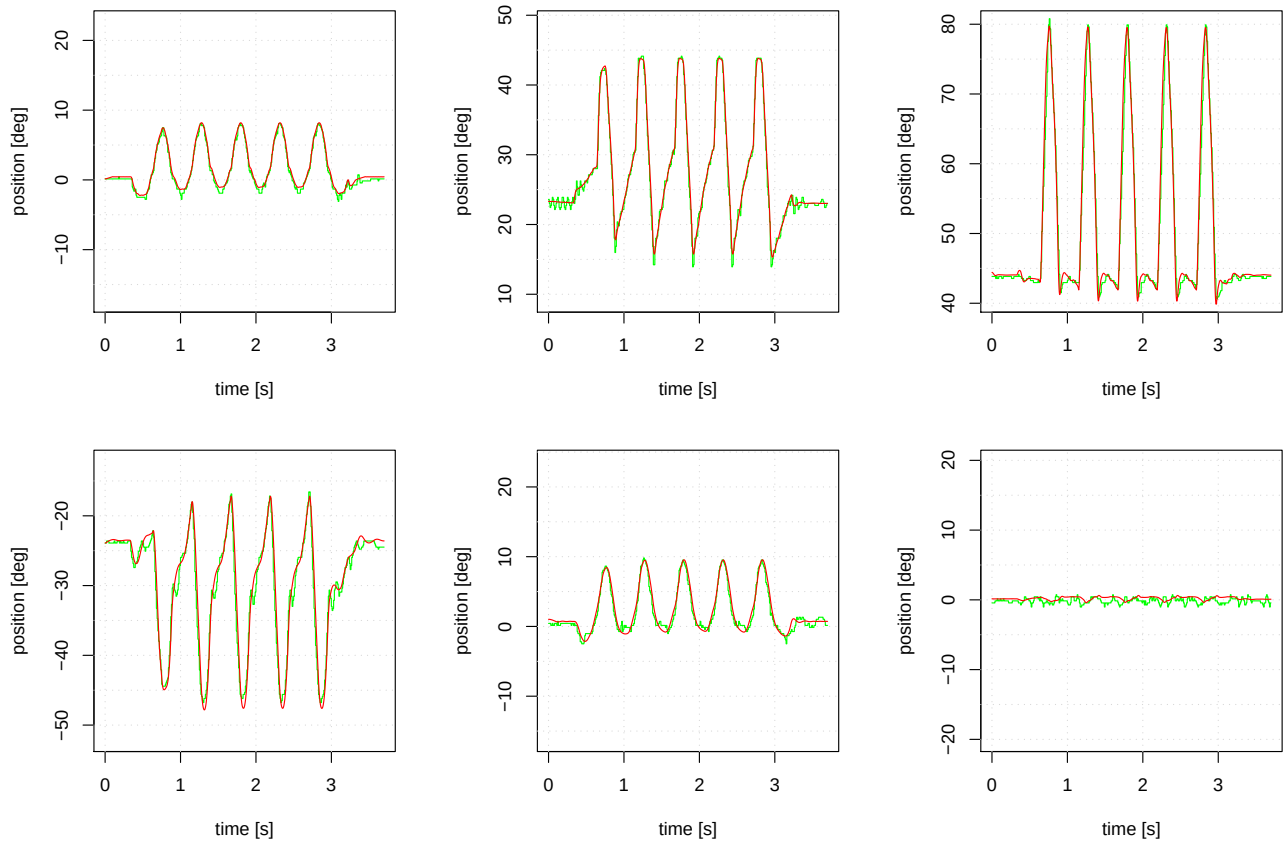


Figure 7.14: Joint angle trajectories of real (green) and simulated (red) robot for the motors of one leg over a period of five strides. The trajectories are given beginning from the foot (upper left) up to the hip (lower right).

the possibility of choosing the level of abstraction for each simulated robot individually. By providing several different methods for the motion simulation on different levels of abstraction, the user is enabled to make a tradeoff between the accuracy and the performance of the simulation. To allow for a qualified decision, all simulation methods have been validated.

7.2 Simulation of a Search-and-Rescue-Robot

In this section a simulation of autonomous robots used for search and rescue applications is presented. To allow for teleoperation without eye-contact as well as for autonomous or semi-autonomous operation, such robots usually are equipped with a wide range of sensors. Besides cameras distance sensors play an important role for this kind of operation as they are well suited for tasks like mapping and collision avoidance. Often several distance sensors like laser scanners or ultrasound sensors are attached to one vehicle.

Simulating many distance sensors though, may become a bottleneck for the simulation's performance. To reduce this problem, the simulation makes use of different methods for the simulation of distance sensors provided in `MuRoSimF`. These methods differ significantly in their performance depending on the number of rays as well as depending on the platform the simulation is executed on. As `MuRoSimF` allows the transparent exchange of the simulation method for each sensor individually, the simulation can be adapted to the computer it is ran on optimally.

7.2.1 The Search-and-Rescue Robot Platform

The *Darmstadt Rescue Robot Team* [18] has been participating in the RoboCup-Rescue league since 2009. The robot used by this team is a ground vehicle equipped with several internal and external sensors connected to two computers. Development of the control software for this robot is supported by a simulation based on `MuRoSimF` [60, 115].

As described in [143, 18] the rescue robot platform consists of a modified RC toy car. The basic motion capabilities of the car have been extended from a two-wheel steering drive to a four-wheel steering drive. The completely modified vehicle consists of the *basic vehicle* and an additional (removable) extension, the so called *vision box*.

The Basic Vehicle

To enable autonomous driving of the vehicle, a microcontroller unit (MCU) and an embedded PC are used. The MCU is used to control the servo-motor used for steering and the motor driver. Further on it provides readings of the internal sensor of the vehicle, namely four wheel encoders, three ultrasound distance sensors and an inertial navigation unit with 3D acceleration sensor and 3D gyroscope. The embedded PC is interfaced to the MCU using an ISA bus based interface. An Hokuyo URG04-LX laser scanner is used as the main external sensor of the basic vehicle. It is connected to the embedded PC using RS232. The laser scanner is attached to the chassis of the vehicle by a tilt servo-motor which is connected to the embedded PC by an RS485 connection.

The Vision Box

The vision box was added to the basic vehicle to provide the computing power and sensors necessary for advanced computer vision and navigation tasks for autonomous rescue vehicles. It consists of a second PC, a normal camera for daylight vision, an infrared camera for thermal vision and an Hokuyo UTM-30LX laser scanner. The cameras are attached to a common pan-tilt-unit, the laser scanner can be moved by a roll-tilt-unit. All servo-motors are connected to the vision PC by RS485.

The vision box has been designed to be easily removable. Besides the mechanical connection and the power connector, only an ethernet line is used for communication between the PC of the vision box and the PC of the basic vehicle.

Structure of the Control Software

Two distinct applications are running on the embedded PC of the basic vehicle: a real-time application directly interfacing the MCU for motion control and a high-level control application for application dependent tasks (e. g. mapping, navigation or remotely controlled driving). The real time application is based on the Xenomai [15, 136] extension for Linux. The high level application is based on the `RoboFrame` framework [116, 118]. Both applications communicate using a serial connection based on named pipes provided by Xenomai.

The application running on the vision box is also based on `RoboFrame`. As `RoboFrame` provides transparent mechanisms for the data exchange between modules running in separate applications, sensor processing for each sensor can be done in each application, independent of the application the sensor is connected to. Only modules directly connected to a hardware device must be executed on the PC connected to the device.

7.2.2 Structure of the Simulation

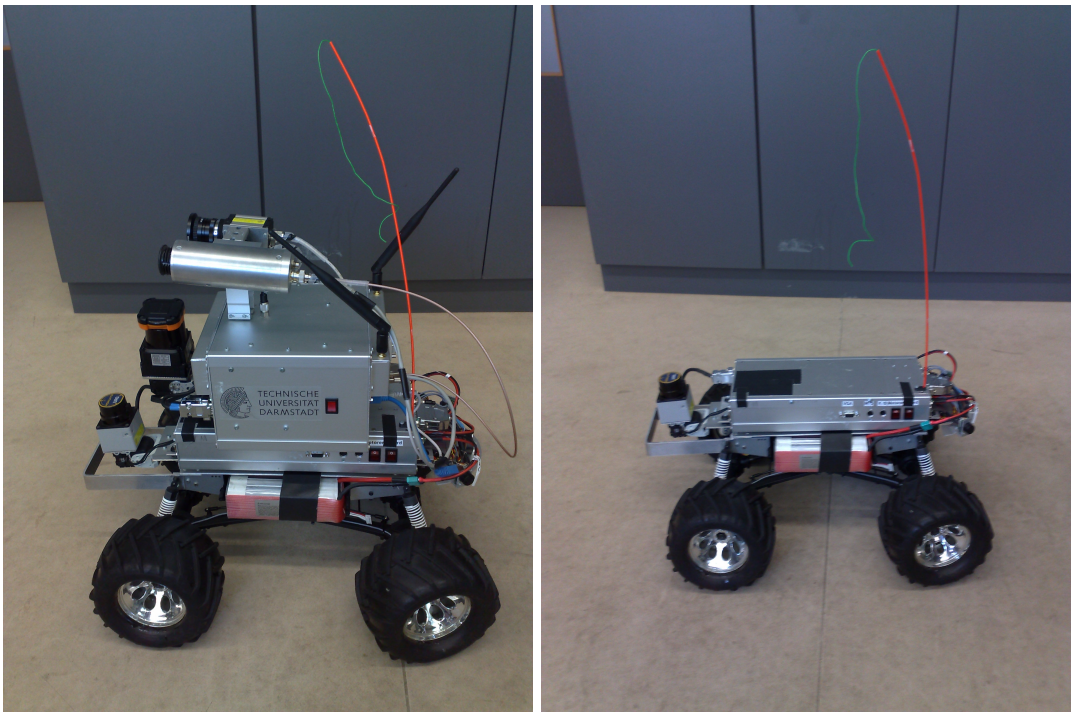
The simulation for the rescue vehicle provides simulation of the robot's motion capabilities as well as for the robot's sensors. Models are created for each simulated vehicle as well as one model for the static environment.

Several simulation tasks are connected to these models: The simulation uses one common collision detection for all data objects. Information provided by this task is used as input for the motion simulation. Additionally the ray-based simulation for distance sensors uses the bounding-volume hierarchy calculated at every timestep of the collision detection for speedup.

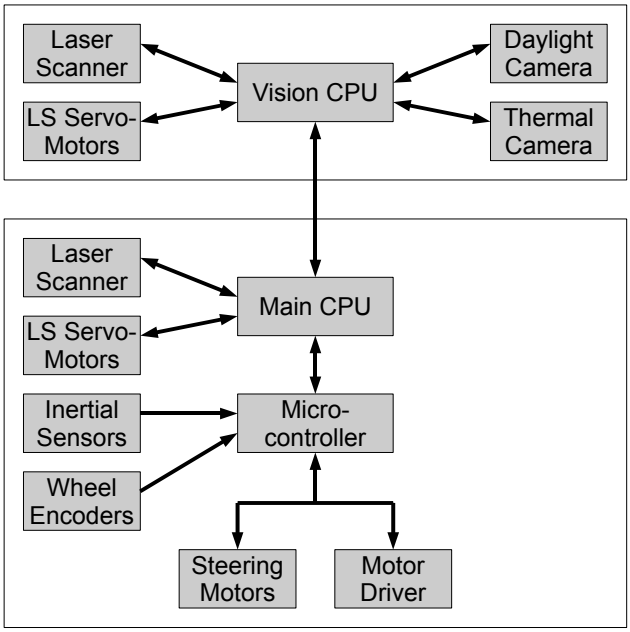
For each robot a set of simulation tasks are used:

- Motion Simulation for the robot is based on the simplified dynamics algorithm. Thus the inner motions of the robot, which are used to move the external sensors, are simulated kinematically while the robot's motion within the environment is based on a dynamical model. Additionally a kinematical simulation of the robots motion is provided which requires less CPU time, but is limited to motion in a plane.

  To interface the robot's motion simulation with the control application, a controller task is provided. This controller uses the same protocol as the real-time layer of the real robot's control software. It is connected by a (virtual) serial connection to the control software,



(a) (b)



(c)

Figure 7.15: The robot of the *Darmstadt Rescue Robot Team*: Images of the robot with (a) and without (b) vision box, (c) hardware structure of the robot.

thus allowing transparent exchange (the control software only has to open another serial device).

- Simulation of distance sensors is provided for both laser scanners of the robot. Two distinct simulation methods can be used, either simulation based on calculation of ray intersections or simulation based on the depth-buffer. It is possible to select the simulation method individually for each of the sensors.

  For each laser scanner a controller task is used. This controller task allows communication with the control application via serial connection. To allow transparent exchange in the controller the SCIP-protocol[2] which is also used to control the real sensor has been implemented.

- Camera simulation is based on real-time rendering. As in the current application the camera images only are used by a human operator, no special efforts were taken to calibrate the camera's significant distortion. Unlike the other controllers the camera simulation cannot be interfaced transparently with the control-application. Instead a TCP connection is used, requiring the exchange of one module in the control-application. Currently no simulation for the infrared camera is provided.

- Internal sensors like the gyroscopes and the acceleration sensors as well as the wheel encoders are simulated based on the values calculated by the motion simulation. The values of the simulated sensors can be accessed using the same protocol as on the real robot through the communication interface also used for the control of the robot's motion. Through the same interface the real robot provides navigation information on the robot's position and orientation. In the simulation these data are replaced by ground truth data.

The execution rate of the collision detection task and the motion simulation of all robots has to be the same. In experiments it has been determined that 200 Hz is sufficient to provide a stable simulation. The execution rates for the tasks concerned with sensor simulation may be chosen individually for each simulated sensor (including switching off individual sensors).

7.2.3 Applications

The simulation has been used during the development of several modules of the control software for the search-and-rescue-robot (see Figure 7.16 and 7.17). Main applications were the testing of the robot's behavior control and mapping capabilities [90]. Further on, an early version of the simulation has been used to evaluate approaches for multi robot coordination[84].

7.2.4 Performance of the Laser Scanner Simulation

The simulation presented in this simulation provides two different methods for the simulation of laser-scanners at any resolution. To evaluate the performance of either method a scenario with four robots was used. The maximum number of scans per second per robot which could be simulated in realtime was measured for several resolutions (see Table 7.9). Measurements were taken on two different computers (see Table 7.1).

Notes according the measurements:

[2] The "Scanning sensor Command Interface Protocol" [85, 4].

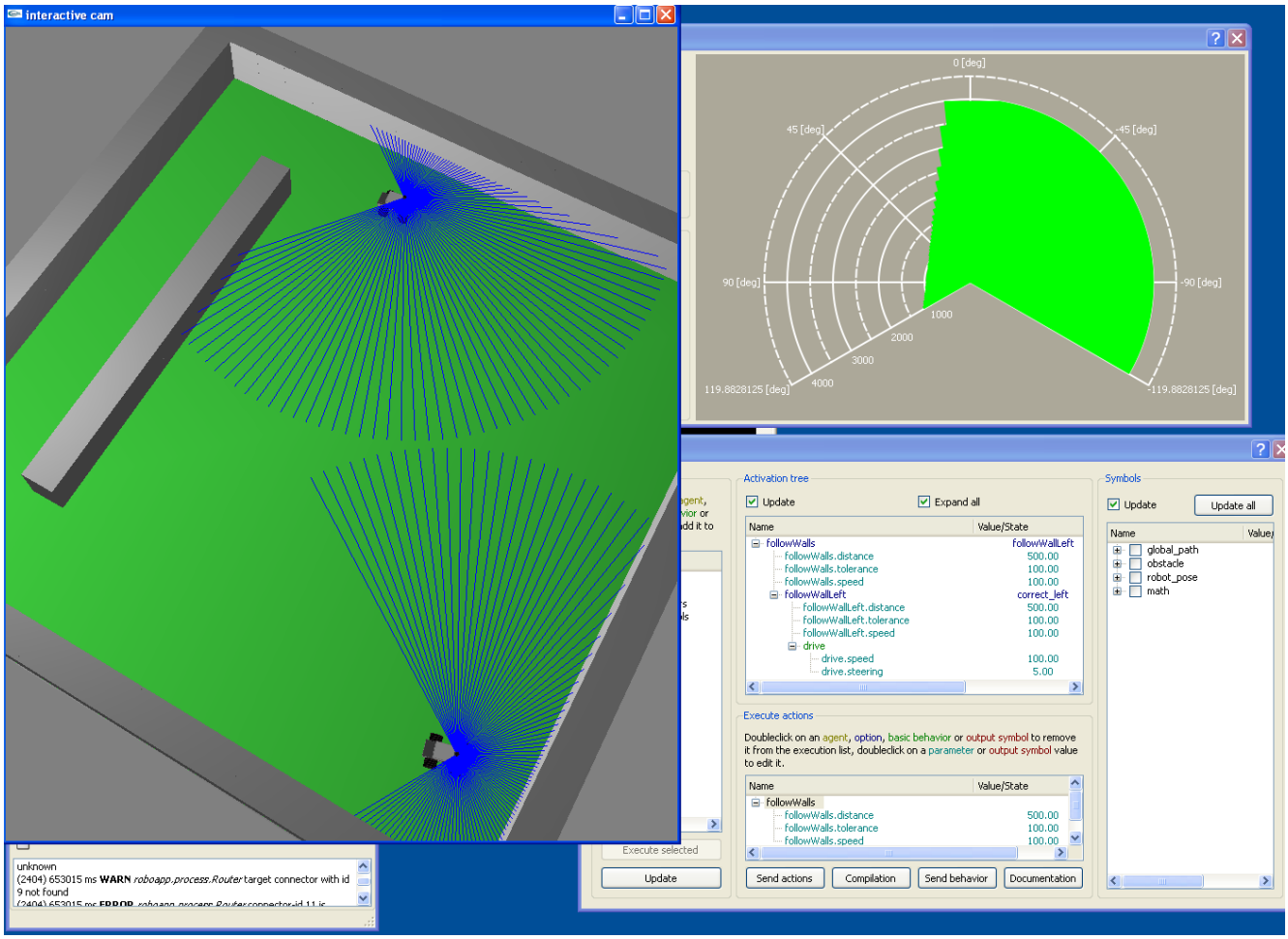


Figure 7.16: Testing of basic sensing and behaviors in a scenario with two autonomous vehicles. Each of the vehicle is controlled by its own control application running a behavior for following walls. The upper right window shows the current scan of the upper vehicle's laser scanner, the lower right window is a display to the vehicle's behavior.

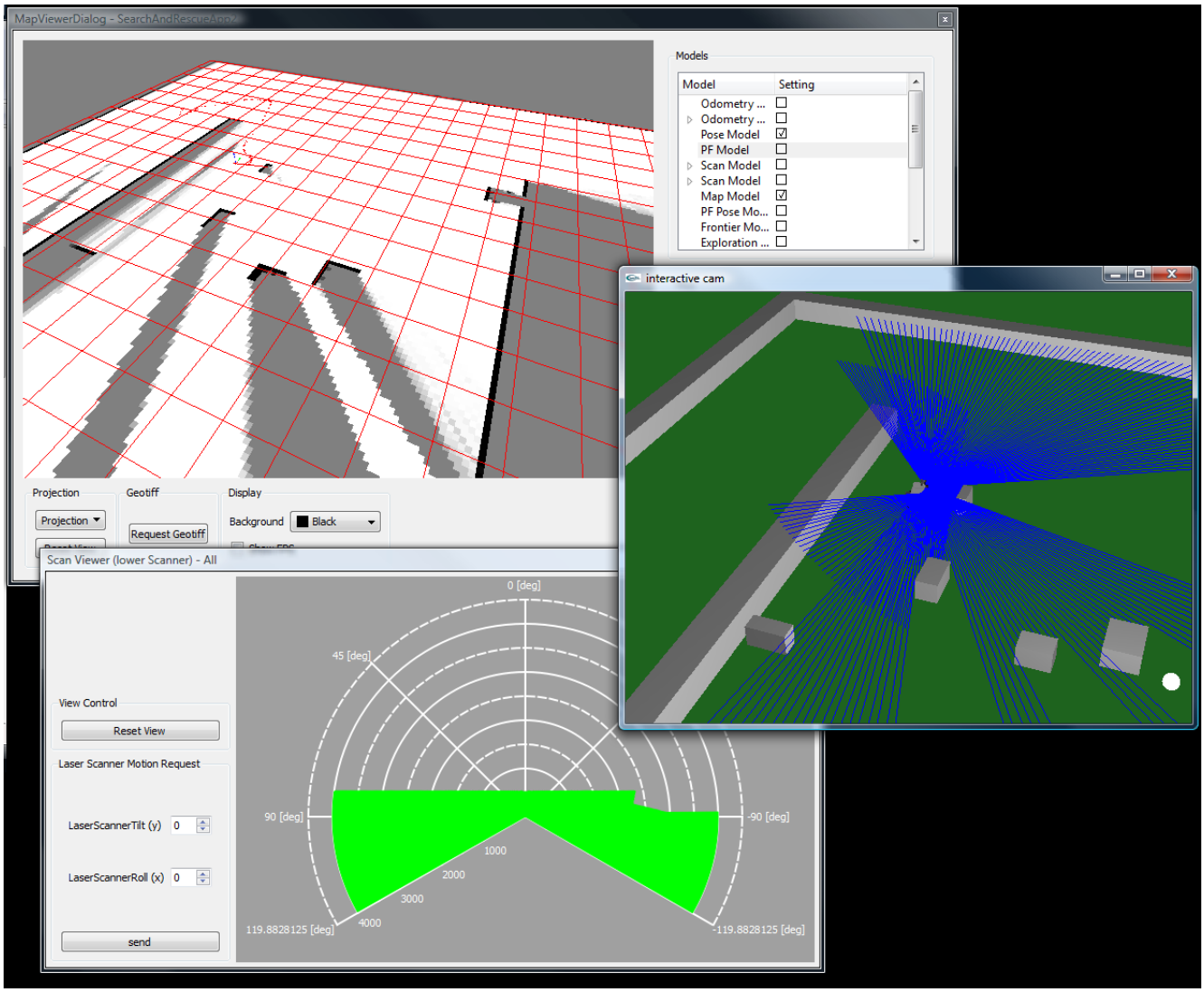


Figure 7.17: Testing the search-and-rescue robot's control application. Upper left: Map created by control application, lower left: visualization of one of the laser scanners, right: interactive simulation of the vehicle (augmented with rays of the two laser scanners.)

Table 7.9: Performance of laser-scanner-simulation. The measured value is the maximum number of scans per second which could be simulated on each of the four robots simultaneously in realtime. R1: single threaded ray intersection, R2: ray intersection with two threads, ZB: z-buffer.

	Computer					
	A			B		
Resolution	R1	R2	ZB	R1	R2	ZB
10 x 1	~ 2700	~ 4500	~ 73	~ 1300	~ 2300	~ 100
50 x 1	~ 580	~ 900	~ 62	~ 280	~ 550	~ 91
100 x 1	~ 300	~ 470	~ 62	~ 150	~ 280	~ 88
500 x 1	~ 60	~ 95	~ 60	30	~ 57	~ 70
1000 x 1	31	50	~ 56	14	26	~ 54
50 x 50	7	14	40	3	6	25
100 x 100	1	2	15	/	/	/

- The opening angle of the laser scanner was 90°, thus allowing the use of only one rendering pass for the depth buffer method. Depending on the opening angle of the laser scanner and additional requirements on the distribution of the rays, it may become necessary to use more rendering passes, thus reducing the number of scans which can be simulated per second.

- As the collision detection and motion simulation modules of the simulation is running at 200 Hz, the ray intersection method will become inaccurate for higher numbers of scans per second.

The following results concerning the performance of the simulation methods can be drawn:

- In general, the simulation method based on reading the depth buffer performs better for higher resolutions, while the method based on ray intersection has a better performance for low resolutions.

- The break even of the two methods depends on the concrete computer. On the newer system A, it is at a higher resolution than on the older system B.

- The method based on ray intersection scales well with the number of threads. For high resolutions (and thus low numbers of scans per second), the additional costs of thread switching and synchronization are minimal, so that the number of scans can be doubled by using two threads.

These results show, that it is very useful to have the possibility to chose from different simulation methods for the same purpose. Depending on the used hardware, the method with the better performance may be chosen, if other criteria (like the differences of both methods concerning the ray distribution and simulation accuracy described in Section 5.4.3) are of no interest.

It should be noted, that the methods for simulation of laser scanners can also be used for other distance sensors like ultrasound. Thus the results presented here are valid for these sensors, too.

7.2.5 Summary

In this section a simulation for a wheeled search-and-rescue robot based on MuRoSimF has been presented. An unique feature of this simulation is the flexible exchange of simulation methods for the robot's laser-scanners allowing to choose the method with the (computer dependent) best performance. Further on the simulation provides access to the simulated drives and laser scanners in the same way as the real robot, thus allowing a transparent exchange of real and simulated robots. The simulation has been used successfully in the development of the robot's basic software, behavior control and mapping algorithms.

7.3 Simulation of a Team of Heterogeneous Robots.

In [86, 87] the cooperation of a strongly heterogeneous team of a wheeled robot (in this case a modified Pioneer 2DX) and a humanoid robot (the 2006 version of the Darmstadt Dribbler's humanoid robot, see [56] and Section 7.1.1) has been investigated. The task for the robots was to jointly follow an object (in this case a ball) over a distance combining their specific capabilities: the humanoid robot provides external sensors capable of recognizing the ball, the wheeled robot provides faster locomotion.

Both robots use a distributed computing system consisting of a microcontroller unit providing basic motion capabilities and an additional computer connected by RS232 for high level control. The high level applications for both robots are based on RoboFrame.

To support development of the high level control software a simulation based on MuRoSimF has been developed. The simulation relies on two distinct features of MuRoSimF: the ability to use different simulation methods for each robot in the same simulation and the flexible controller concept allowing the integration of robot specific controllers. Motion simulation for both robots is based on specialized kinematic algorithms (biped walking simulation for the humanoid robot, differential drive for the wheeled robot). The only external sensors simulated are the cameras of the humanoid robot. For each of the robots a controller is simulated providing the same communication protocol as the real robot, thus allowing easy transfer of the the high level control software from simulation to the real robot.

Experiments described in [86, 87] were conducted on the real robots as well as in the simulation and showed comparable results. Software developed with the simulation could be transfered to the real robots.

7.4 Further Application Examples

7.4.1 Prototyping of a New Four Legged Robot Platform

To support the development of a new four legged robot [63, 64], a simulation based on MuRoSimF was created (see Figure 7.19). The robot has 15 degrees of freedom (3 per leg, 3 in the neck). It is equipped with a camera in the head and several other sensors. Computation is mainly performed on an embedded PC, but an microcontroller (connected via RS232) is used to interface to the servo-motors and to the sensors.

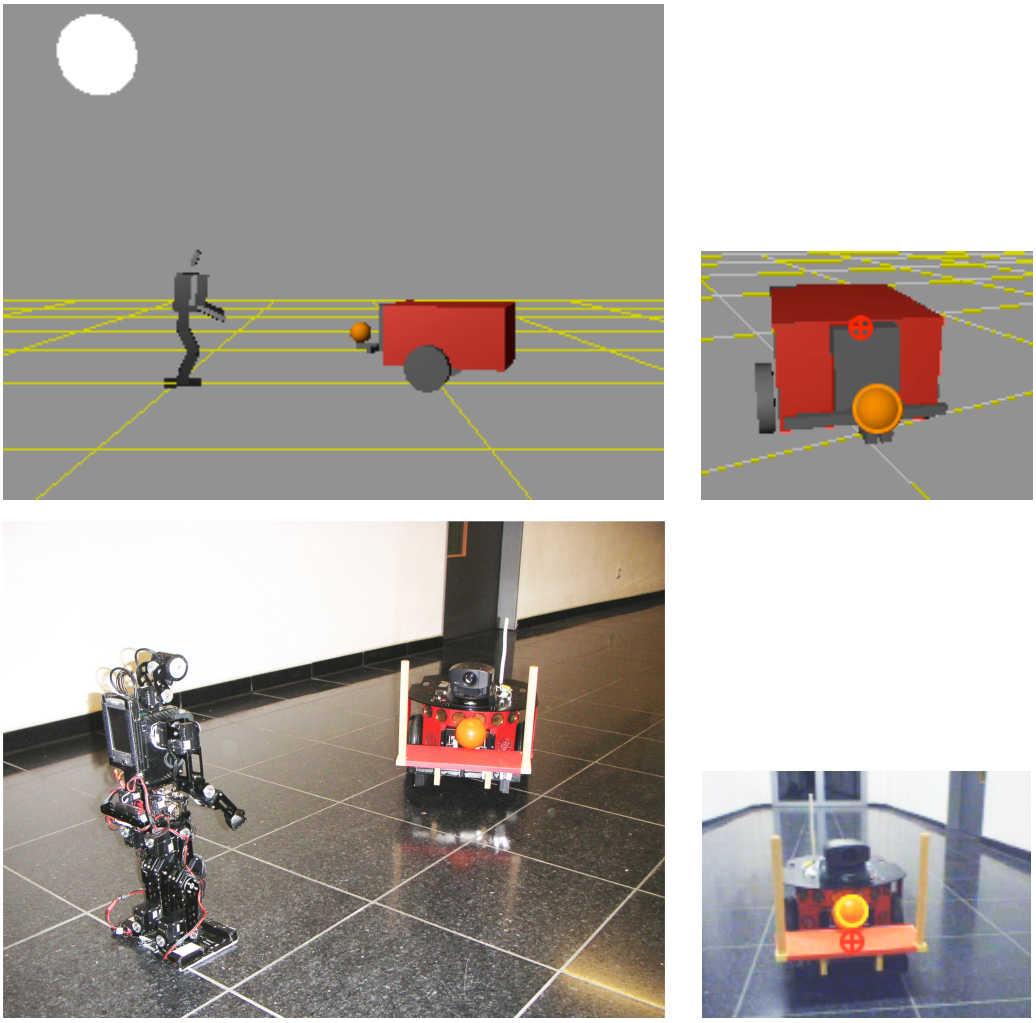


Figure 7.18: Simulation of a team of heterogeneous robots. Upper left: Simulated scenario. Upper right: simulated camera. Lower left: Real scenario. Lower right: real camera. Note that both camera images are augmented with percept-informations generated by the vision module of the humanoid robot's control applicaitons. Images taken from [86].

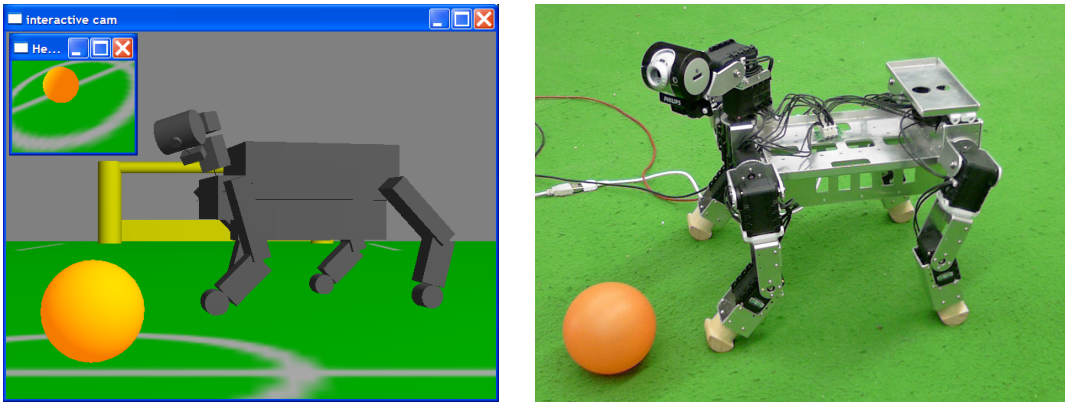


Figure 7.19: Simulated and real prototype of a four legged robot[64]. Left: Simulation of the four legged robot (including camera in the small window). Right: Prototype of the robot.

A simulation is provided for the robot's motion apparatus (based on the simplified dynamics approach) and the camera. Additionally a controller module is provided which can be used to connect the simulation to the high level control application.

The simulation has been used in several phases of the design process. During mechanical design, it has been used to evaluate the kinematic design of the robot's head in order to ensure versatile viewing capabilities for all gaits of the robot. While the first prototype of the robot was being manufactured, the simulation was used to develop the communication protocol for the microcontroller as well as a prototype motion engine providing walking patterns for the robot. The basic motion engine later on could be transfered to the real robot, but the parameters describing the robot's gaits had to be adapted, as the real servo-motors behaved differently from the simulated ones.

7.4.2 Collision Detection for a Pipe-Bending Robot

For an industrial partner a simulation of a pipe bending robot has been developed. Purpose of the simulation is the detection of collisions between the pipe on the one hand and the bending robot and environment on the other (see Figure 7.20). The simulation is not used as a stand-alone program but as a module embedded within the control software for the robot. From there it is used during the derivation and optimization of bending programs after a pipe has been specified.

Input data for the simulation is a description of the machine (kinematic structure and shapes described as triangular meshes) and a description of the bending program and the pipe. Output data are collisions detected during the bending process with information on the time of the program, the position of the collision and the parts involved in the collision.

The main modules of the simulation are:

- Calculation of the motion of the machine: For this module the existing forward kinematics implementation could be reused without changes, thus saving development and testing time.

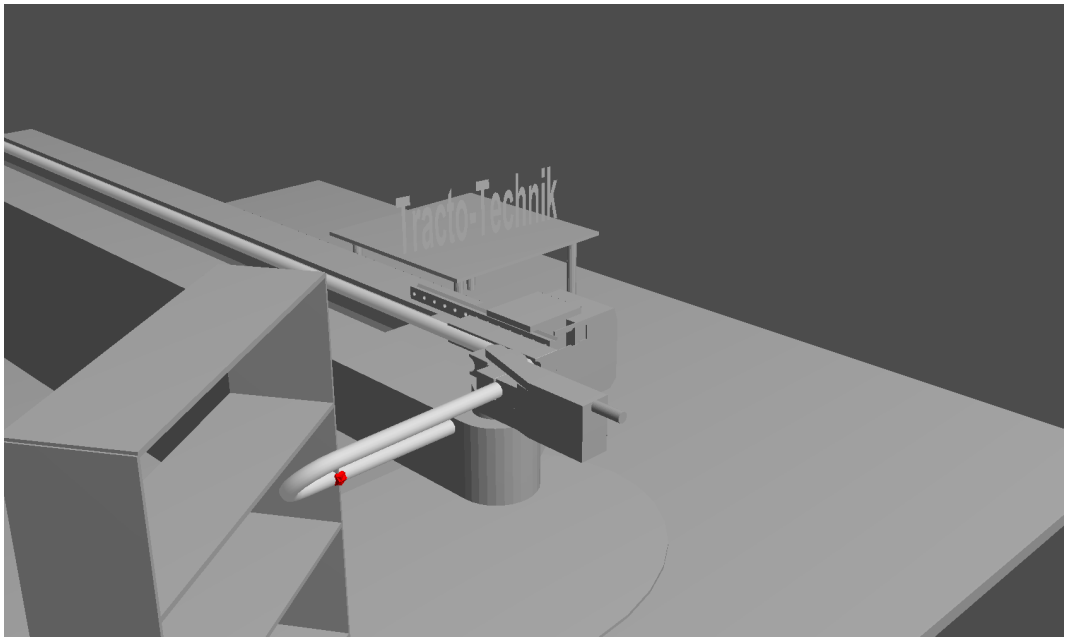


Figure 7.20: Simulation of a pipe-bending robot. Detected collisions between the pipe and the shelf on the left are marked with red points.

- Calculation of the motion of the pipe: A new algorithm based on the forward kinematics calculation had to be developed which considers the fact that the pipe changes its shape during the bending process.
- Collision detection between pipe and other objects: The collision detection is the central part of the simulation. It is based on the general collision-detection module provided by `MuRoSimF` reusing existing algorithms for bounding-volume tests and selection of bodies for further testing. An application specific `CollisionDetector` was developed. It uses a hybrid collision-detection between objects described by triangular meshes and segments of the pipe described as parametric surfaces. As meshes and pipe-segments alike may be concave, the hybrid approach is much more efficient than a collision detection based only on meshes.
- Step width control.
- Customer specific input and output modules for model data and detected collisions.

Development of the simulation was based on `MuRoSimF`. Significant parts of the software could be realized using existing classes. Application specific additions (a special shape and an optimized collision-detection-algorithm) could be easily integrated with the existing framework. During debugging of the simulation the existing visualization components of `MuRoSimF` could be used.

8 Conclusions

The main topic of this thesis is the simulation of teams of autonomous robots with different levels of abstraction. As motivated in Chapter 1 it is desirable to simulate autonomous robots with different levels of abstraction depending on the concrete application of the simulation. An analysis in Chapter 2 reveals, that current simulations for autonomous robots only provide very limited support for multiple levels of abstraction. Especially simulation of robot motion is limited in most simulations to one distinct level often based on a single, not exchangeable simulation method. Only few robot simulations provide the means for exchanging simulation methods, but those are limited to methods on the same level of abstraction or to methods for very specific purposes. Thus the user is forced to use different simulation programs if different levels of abstraction are required.

In Chapter 3 a methodology for simulations with adaptable levels of abstraction is proposed. This methodology is the base for the Multi Robot Simulation Framework (`MuRoSimF`) which has been developed as part of this thesis. In Chapter 4 a modeling approach for autonomous robots is developed which supports the transparent exchange of simulation methods on different levels of abstraction. A wide range of methods for the simulation of motion and sensors of autonomous robots on several levels of abstraction is presented in Chapter 5. The integration of models and simulation methods into executable simulations are covered in Chapter 6. In Chapter 7 simulations are presented which have been created using the proposed methodology and `MuRoSimF`.

The methodology and simulation methods developed in this thesis allow a flexible configuration of simulations of autonomous robots. It is possible to combine several simulation methods on different levels of abstraction within the same simulation and to use these methods simultaneously. Due to this flexibility provided by `MuRoSimF` robot simulations can now be created which can not just be used for different tasks, but can be configured to provide an adequate level of abstraction for each of these tasks.

If several simulation methods need to be used in concert to achieve a certain purpose, each method can be chosen independently, allowing for a flexible configuration of the simulation. An application of this feature of `MuRoSimF` can be found in sensor simulation. When simulating a sensor for a certain physical property, at least two simulation methods must be incorporated, one providing the value of the property and one simulating the sensing process. For either purpose different methods may be used, e. g. physical simulation or simplified approximations for providing the value and sensing with or without simulated errors for the sensor simulation. In contrast to this in most existing standard robot simulation programs both parts are coupled, as the methods for simulating the sensing-part directly access the part responsible for the physical simulation, so that an exchange of the physical simulation unavoidably results in subsequent changes to the sensor simulation.

Several simulations for different kinds of robots and with different levels of abstraction have been developed:

The humanoid robot simulation developed in this thesis provides several different levels of abstraction for the 3D motion simulation of a team of robots with kinematical or dynamical methods. As a unique feature it is not just possible to select one method for the whole simulation, but instead the level of abstraction may be chosen for each of the robots individually.

Each of the simulation methods has been validated to provide a guideline for the selection of method adequate for a given task. While providing different simulation methods for the robots' motions, the robots always can be interfaced from the control software in the same way, which is a transparent replacement for the communication with the real robot.

The simulation for search and rescue robots features transparent exchange of the methods for the simulation of distance sensors. While the provided methods do not differ in what they simulate, they strongly differ in the way the simulation is done and thus in the performance of the method. Depending on the computer the simulation is running on as well as on the resolution of the respective sensor, either method may be of best performance. Due to the ability of choosing the method for each simulated sensor individually and transparently, it is possible to use the combination of methods which perform best on the respective computer.

In the simulation for a team of heterogeneous robots, two robot specific kinematic methods for motion simulation are combined providing an adequate level of abstraction for the research on team cooperation done with this simulation.

The simulation of a pipe bending robot reuses methods for the kinematic simulation of the robot's motion and combines them with a domain specific collision detection adequate to the given task.

These application examples demonstrate the versatility of the robot simulation methodology developed in this thesis. All simulations presented in this thesis were created using `MuRoSimF`. Each of them provides the levels of abstraction which are adequate to the given task.

9 Zusammenfassung (Abstract in German)

Eines der wichtigsten Werkzeuge bei der Entwicklung von Soft- und Hardware für autonome mobile Roboter ist deren Simulation. Die Simulation solcher Roboter betrachtet dabei üblicherweise sowohl die Sensorik und den Bewegungsapparat der Roboter als auch ihre Interaktion mit der Umgebung. Abhängig vom Einsatz der Simulation werden verschiedene Abstraktionsgrade benötigt, z. B. Dynamiksimulation der Bewegungen des Roboters im Raum oder vereinfachte Simulationen, die nur die Bewegung des Gesamtsystems in der Ebene betrachten. Bestehende Simulationsprogramme sind dabei stark eingeschränkt bezüglich der Verwendung verschiedener Abstraktionsgrade. Häufig wird genau ein Abstraktionsgrad angeboten, Möglichkeiten zur Integration alternativer Simulationsmethoden bestehen kaum. Werden mehrere Abstraktionsgrade benötigt, ist es bei Verwendung bestehender Programme notwendig, mehrere verschiedene Simulationen einzusetzen. Dies erhöht jedoch sowohl den Arbeitsaufwand als auch die Anzahl möglicher Fehlerquellen bei der mehrfachen Modellierung der Roboter und der Anbindung mehrerer Simulationen an die jeweilige Steuerungssoftware der Roboter.

Die vorliegende Arbeit beschäftigt sich mit der Simulation autonomer Roboterteams unter der Berücksichtigung verschiedener Abstraktionsgrade für die Simulation. Ziel ist es, Simulationsprogramme zu erstellen, die es gestatten, Simulationsmethoden und -modelle für verschiedene Abstraktionsgrade innerhalb eines Simulationsprogramms flexibel kombinieren und austauschen zu können. Damit soll es ermöglicht werden, aus einem Programm heraus mehrere Abstraktionsgrade anzubieten, so dass stets der adäquaten Abstraktionsgrad für eine konkrete Anwendung ausgewählt werden kann.

Ausgehend von einer Analyse verschiedener Abstraktionsgrade für die Simulation der Sensorik und des Bewegungsapparates autonomer mobiler Roboter wird eine neue Methodik entwickelt, die es erlaubt, Simulationen zu erstellen, die unterschiedliche Abstraktionsgrade flexibel vereinigen. Ein Kernpunkt der Methodik ist dabei die Modellierung der simulierten Roboter auf eine Art, die es ermöglicht verschiedenste Simulationsmethoden, auch gleichzeitig, zum Einsatz zu bringen. Sowohl das Modellierungskonzept als auch die Anbindung der Simulationsmethoden ist dabei auf flexible und transparente Erweiterbarkeit ausgelegt. Auch wenn beim gemeinsamen Einsatz mehrerer Simulationsmethoden Abhängigkeiten bezüglich der berechneten Informationen bestehen, ist es möglich, jede der Methoden individuell gegen eine andere auszutauschen, solange die selben Informationen, ggf. auf einem anderen Abstraktionsgrad, bereitgestellt werden. Ein weiterer Aspekt der entwickelten Methodik ist die Validierung von Simulationen mit unterschiedlichem Abstraktionsgrad. Dieser Validierung kommt eine besondere Bedeutung zu, da sie die Grundlage für eine begründete Entscheidung für oder gegen eine bestimmte Simulationsmethode aus einer Menge verfügbarer Methoden darstellt.

Die entwickelte Methodik bildet die Grundlage für das Multi-Robot-Simulation-Framework (`MuRoSimF`), das im Rahmen dieser Arbeit erstellt worden ist. In `MuRoSimF` werden eine Vielzahl verschiedener Methoden zur Simulation der Sensorik und des Bewegungsapparates mobiler autonomer Roboter mit verschiedenen Abstraktionsgraden bereitgestellt und können flexibel kombiniert werden.

Mit `MuRoSimF` entwickelte Simulationen sind dabei in der Lage, Roboter mit unterschiedlichen Antriebskonzepten, z. B. Humanoidroboter, Fahrzeuge oder vierbeinige Roboter sowohl mit allgemeinen als auch mit antriebs-spezifischen Methoden zu simulieren. Im Gegensatz zu her-

kömmlichen Simulationsprogrammen ist es dabei möglich, verschiedene Simulationsmethoden flexibel zu kombinieren und gleichzeitig anzuwenden. Dies zeigt sich in verschiedenen in dieser Arbeit beschriebenen Simulationen. So gestattet es z. B. die beschriebene Simulation für Teams von Humanoidrobotern, für jeden der betrachteten Roboter das Verfahren zur Simulation der Bewegung des Roboters individuell zu wählen. Damit wird es möglich, einen der Roboter genauer zu simulieren, während er auf die Aktionen andere Roboter reagiert (die zu diesem Zweck mit geringerem Detailgrad simuliert werden können). Ein anderes aufgeführtes Beispiel ist die Simulation eines heterogenen Team aus einem humanoiden und einem radgetriebenen Roboter. Zweck der Simulation ist die Untersuchung der Aufgabenverteilung und Kooperation in heterogenen Teams. Da in diesem Fall keine komplexe Dynamiksimulation der beteiligten Roboter benötigt wird, werden zwei roboter-spezifische Methoden kombiniert, um eine für den Anwendungsfall adäquate Simulation bereitzustellen.

Obwohl die beschriebene Methodik primär für die Simulation autonomer mobiler Roboter entwickelt wurde, ist sie nicht auf diese beschränkt. Dies wird durch die in dieser Arbeit beschriebene Simulation eines Industrieroboters, die zum Zweck der Planung kollisionsfreier Bahnen für das Werkstück entwickelt wurde, unterstrichen. In dieser Simulation wird eine für die Anwendung angepasste, spezialisierte Kollisionserkennung mit bestehenden Verfahren zur Kinematiksimulation kombiniert.

Durch den Einsatz der in dieser Arbeit entwickelten Methodik ist es möglich, Simulationen für autonome Roboter zu erstellen, die verschiedene Abstraktionsgrade bereitstellen. Dabei wird eine Flexibilität erreicht, die mit herkömmlichen Roboter-Simulationen nicht möglich ist. Durch das modulare Modellierungskonzept ist eine einfache Erweiterbarkeit bereits in `MuRoSimF` vorhandener Simulationsmethoden für neue Anwendungen, wie z. B. die Simulation von Robotern mit elastischen Elementen oder biomechanischer Systeme, sichergestellt.

Bibliography

[1] Blender For Robotics – http://wiki.blender.org/index.php/Robotics:Index.

[2] OpenRobots Simulator – http://wiki.blender.org/index.php/Robotics:Simulators/OpenRobots.

[3] MathEngine Karma(TM) User Guide (Version 1.2). Technical report, MathEngine, 2002.

[4] URG Series Communication Protocol Specification SCIP-Version 2.0. Technical report, Hokuyo Automatic co., Ltd., 2006.

[5] Microsoft Robotics Studio, msdn.microsoft.com/robotics/, 2007.

[6] Newton website, http://www.newtondynamics.com/, 2007.

[7] NVIDIA PhysX website, http://developer.nvidia.com/object/physx.html, 2007.

[8] Epic Games, Unreal Engine, www.unrealtechnology.com, 2008.

[9] GIMPACT Geometric tools for VR, website http://gimpact.sourceforge.net, 2008.

[10] Bullet Physics Library, website http://bulletphysics.com, 2009.

[11] OpenHRP, website http://www.openrtp.jp/openhrp3/en/index.html, 2009.

[12] Persistence of Vision Raytracer (POVRay), website: www.povray.org, 2009.

[13] Simbad website: http://simbad.sourceforge.net/, 2009.

[14] Webots user guide 6.1.5. Technical report, Cyberbotics, Ltd., 2009.

[15] Xenomai: Real-Time Framework for Linux, www.xenomai.org, 2009.

[16] R. Adobbati, A. N. Marshall, A. Scholer, S. Tejada, G. A. Kaminka, S. Schaffer, and C. Sollitto. Gamebots: a 3D Virtual World Test-bed for Multi-Agent Research. In O. Rana and T. Wagner, editors, *Proc. of the 2nd Intl. Workshop on Infrastructure, MAS and MAS Scalability*, May 2001.

[17] A. Albu-Schäffer. *Regelung von Robotern mit elastischen Gelenken am Beispiel der DLR-Leichtbauarme*. PhD thesis, Lehrstuhl für Steuerungs- und Regelungstechnik, Technische Universität München, 2001.

[18] M. Andriluka, M. Friedmann, S. Kohlbrecher, J. Meyer, K. Petersen, C. Reinl, P. Schauß, P. Schnitzspan, A. Strobel, D. Thomas, and O. von Stryk. RoboCupRescue 2009 - Robot League Team: Darmstadt Rescue Robot Team (Germany). Technical report, Technische Universität Darmstadt, 2009.

[19] K. Asanuma, K. Umeda, U. Ryuichi, and T. Arai. Development of a Simulation of Environment and Measurement for Autonomous Mobile Robots Considering Camera Characteristics. In D. Polani, B. Browning, A. Bonarini, and Y. K., editors, *RoboCup 2003, Robot Soccer World Cup VII*, volume 3020 of *LNAI*, pages 446–457. Springer, 2004.

[20] B. Balaguer, S. Balakirsky, S. Carpin, M. Lewis, and C. Scrapper. USARsim: a validated simulator for reserach in robot and automation. In *Workshop on "Robot simulators: available software, scientific applications and future trends" at IEEE/RSJ IROS 2008*, 2008.

[21] B. Balaguer and S. Carpin. Where am I? A Simulated GPS Sensor for Outdoor Robotic Applications. In S. C. et al., editor, *Proc. of the 1st Intl. Conf. on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR)*, LNAI, pages 222–233, Venice, Italy, Nov 4-6 2008. Springer.

[22] D. Baraff. Fast Contact Force Computation for Nonpenetrating Rigid Bodies. In *SIGGRAPH 94: Proc. of the 21st annual conference on computer graphics and interactive techniques*, 1994.

[23] D. Baraff. Linear Time Dynamics using Lagrange Multipliers. In *SIGGRAPH 96: Proc. of the 23rd annual conference on computer graphics and interactive techniques*, 1996.

[24] D. Baraff and A. Witkin. Physically Based Modeling: Principles and Practice, 1997.

[25] R. Barzel and B. A. H. A Modeling System Based On Dynamic Constraints. In *SIGGRAPH 88: Proc. of the 15th annual conference on computer graphics and interactive techniques*, 1988.

[26] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool. SURF: Speeded Up Robust Features. *Computer Vision and Image Understanding (CVIU)*, 110(3):346–359, 2008.

[27] D. Beck, A. Ferrein, and G. Lakemeyer. A Simulation Environment for Middle-size Robot wiht Multi-level Abstraction. In U. Visser, F. Ribeiro, T. Ohashi, and F. Dellaert, editors, *RoboCup 2007: Robot Soccer World Cup XI*, volume 5001, page LNAI. Springer, 2008.

[28] S. Behnke. Online Trajectory Generation for Omnidirectional Biped Walking. In *Proc. of the 2006 IEEE Intl. Conf. on Robotics and Automation (ICRA)*, 2006.

[29] G. A. Bekey. *Autonomous robots: from biological inspiration to implementation and control*. MIT Press, 2005.

[30] J. Bender. Impulse-based dynamic simulation in linear time. *Computer Animation and Virtual Worlds*, 18(4-5):225–233, 2007.

[31] J. Bender, D. Finkenzeller, and A. Schmitt. An impulse-based dynamic simulation system for VR applications. In *Proceedings of Virtual Concept 2005*, Biarritz, France, 2005.

[32] J. Bender and A. Schmitt. Fast Dynamic Simulation of Multi-Body Systems Using Impulses. In *Virtual Reality Interactions and Physical Simulations (VRIPhys)*, pages 81–90, Madrid (Spain), November 2006.

[33] J. Boedecker and M. Asada. SimSpark - Concepts and Applications in the RoboCup 3D Soccer Simulation League. In E. Menegatti, editor, *Workshop Proceedings of SIMPAR 2008, Intl. Conf. on Simulation, Modeling and Programming for Autonomous Robots*, pages 174–181, Venice (Italy), November 3-4 2008.

[34] J. Boedecker, K. Dorer, M. Rollmann, Y. Xu, and F. Xue. SimSpark User's Manual Version 1.1. Technical report, 2008.

[35] J.-Y. Bouguet. Camera Calibration Toolbox for Matlab - website: http://www.vision.caltech.edu/bouguetj/calib_doc/, 2008.

[36] R. A. Brooks. A Robust Layered Control System For A Mobile Robot. *IEEE Jounal of Robotics and Automation*, 2:14–23, 1986.

[37] D. C. Brown. Close-range camera calibration. *PHOTOGRAMMETRIC ENGINEERING*, 37(8):855–866, 1971.

[38] B. Browning and E. Tryzelaar. Übersim: a multi-robot simulator for robot soccer. In *Proc. of the Intl. Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 948–949, 2003.

[39] H. Bruyninckx. Blender for Robotics – Roadmap: http://people.mech.kuleuven.be/˜bruyninc/blender/roadmap.html.

[40] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper. USARSim: a robot simulator for research and education. In *Proc. of the 2007 IEEE Intl. Conf. on Robotics and Automation (ICRA)*, 2007.

[41] S. Carpin, T. Stoyanov, Y. Nevatia, M. Lewis, and J. Wang. Quantitative assessments fo USARSim accuracy. In S. C. et al., editor, *Proc. of the Performace Metrics for Intelligent Systems (PERMIS) Workshop 2006*, volume 5325 of *LNAI*, Venice, Italy, November 2006. Springer.

[42] T. H. J. Collett, B. A. MacDonald, and B. P. Gerkey. Player 2.0: Toward an Practical Robot Programming Framework. In *Proc. of the 2005 Australasian Conference on Robotics and Automation (ACRA)*, 2005.

[43] P. Corke. A Robotics Toolbox for MATLAB. *IEEE Robotics and Automation Magazine*, 3(1):24–32, Mar. 1996.

[44] E. Coumans. Bullet 2.74 Physics SDK Manual. Technical report, 2009.

[45] J. J. Craig. *Robotics*. Addison-Wesley, 1989.

[46] G. Dudek and M. Jenkins. *Computational Principles of Mobile Robots*. Cambridge University Press, 2000.

[47] H. Durrant-Whyte and T. Bailey. Simultaneous Localization and Mapping (SLAM): part I. *Robotics and Automation Magazine*, 13(2):99–110, 2006.

[48] H. Durrant-Whyte and T. Bailey. Simultaneous Localization and Mapping (SLAM): part II. *Robotics and Automation Magazine*, 13(3):108–117, 2006.

[49] C. Ericson. *Real-Time Collision Detection*. Morgan Kaufmann Series in Interactive 3D Technology. Morgan Kaufmann Publishers, 2005.

[50] F. Faber and S. Behnke. Stochastic Optimization of Bipedal Walking using Gyro Feedback and Phase Resetting. In *Proc. of 2007 IEE-RAS Intl. Conf. on Humanoid Robots*, Pittsburgh, PA, USA, Nov. 29 - Dec. 1 2007.

[51] R. Featherstone. The Calculation of Robot Dynamics Using Articulated-Body Inertias. *Intl. J. of Robotics Res.*, 2(1):13–30, 1983.

[52] R. Featherstone. *Rigid Body Dynamics Algorithms*. Springer, 2008.

[53] R. Featherstone and D. Orin. Robot Dynamics: Equations and Algorithms. In *Proc. of the 2000 IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 826–834, San Francisco, CA, USA, April 2000.

[54] J. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice in C (2nd Edition)*. Addison-Wesley, 1995.

[55] D. Fox, W. Burgard, and S. Thrun. Markov Localization for Mobile Robots in Dynamic Environments. *Journal of Artificial Intelligence Research*, 11:391 – 427, 1999.

[56] M. Friedmann, J. Kiener, S. Petters, H. Sakamoto, D. Thomas, and O. von Stryk. Versatile, high-quality motions and behavior control of humanoid robots. *International Journal of Humanoid Robotics*, 5(3):417–436, September 2008.

[57] M. Friedmann, J. Kiener, S. Petters, D. Thomas, and O. von Stryk. Modular software architecture for teams of cooperating, heterogeneous robots. In *Proc. IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 613–618, Kunming, China, December 17-20 2006.

[58] M. Friedmann, J. Kiener, S. Petters, D. Thomas, and O. von Stryk. Reusable architecture and tools for teams of lightweight heterogeneous robots. In *Proc. 1st IFAC Workshop on Multivehicle Systems*, pages 51–56, Salvador, Brazil, Oct. 2-3 2006.

[59] M. Friedmann, J. Kiener, S. Petters, D. Thomas, and O. von Stryk. Darmstadt Dribblers: Team Description for Humanoid KidSize League of RoboCup 2007. Technical report, Technisch Universität Darmstadt, 2007.

[60] M. Friedmann, K. Petersen, and O. von Stryk. Simulation of Multi-Robot Teams with Flexible Level of Detail. In S. C. et al., editor, *Proc. of the 1st Intl. Conf. on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR)*, LNAI, pages 29–40, Venice, Italy, Nov 4-6 2008. Springer.

[61] M. Friedmann, K. Petersen, and O. von Stryk. Tailored real-time simulation for teams of humanoid robots. In U. Visser, F. Ribeiro, T. Ohashi, and F. Dellaert, editors, *RoboCup 2007: Robot Soccer World Cup XI*, volume 5001 of *LNAI*, pages 425–432. Springer, 2008.

[62] M. Friedmann, K. Petersen, and O. von Stryk. Adequate Motion Simulation and Collision Detection for Soccer Playing Humanoid Robots. *Robotics and Autonomous Systems*, 57:786–795, 2009.

[63] M. Friedmann, S. Petters, M. Risler, H. Sakamoto, D. Thomas, and O. von Stryk. A new, open and modular platform for research in autonomous four-legged robots. In K. Berns and T. Luksch, editors, *Autonome Mobile Systeme 2007*, Informatik aktuell, pages 254–260, Kaiserslautern, Oct. 18 - 19 2007. Springer Verlag.

[64] M. Friedmann, S. Petters, M. Risler, H. Sakamoto, D. Thomas, and O. von Stryk. New Autonomous, Four-Legged and Humanoid Robots for Research and Education. In E. Menegatti, editor, *Workshop Proceedings of SIMPAR 2008, Intl. Conf. on Simulation, Modeling and Programming for Autonomous Robots*, pages 570–579, Venice (Italy), Nov. 3-4 2008.

[65] K. Fu, R. Gonzalez, and C. Lee. *Robotics*. McGraw-Hill, 1987.

[66] B. P. Gerkey, R. T. Vaughan, and A. Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *Proc. of the 2003 Intl. Conf. on Advanced Robotics (ICAR)*, pages 317–323, Coimbra, Portugal, 30 June - 3 July 2003.

[67] B. P. Gerkey, R. T. Vaughan, K. Støy, A. Howard, G. S. Sukhtame, and M. J. Matarić. Most Valuable Player: A Robot Device Server for Distributed Control. In *Proc. of the 2001 IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 1226–1231, 2001.

[68] K. Ghaze-Zahedi, T. Laue, T. Röfer, P. Schöll, K. Spiess, A. Twickel, and S. Wischmann. RoSiML - Robot Simulation Markup Language, http://www.informatik.uni-bremen.de/spprobocup/RoSiML.html.

[69] J. Go, B. Browning, and M. Veloso. Accurate and flexible simulation for dynamic, vision-centric robots. In *Pro. of the Intl. Joint Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS)*, 2004.

[70] A. Goswami. Postural Stability of Biped Robots and the Foot-Rotation Indicator (FRI) Point. *The International Journal of Robotics Research*, 18(6):523–533, 1999.

[71] N. Greggio, G. Silvestri, S. Antonello, M. E, and E. Pagello. A 3D Model of a Humanoid Robot for USARSim Simulator. In *Proc. 1st Workshop on Humanoid Soccer Robots at the 2006 IEEE-RAS Intl. Conf. on Humanoid Robots*, pages 17–24, Genua, December 2006.

[72] N. Greggio, G. Silvestri, E. Menegatti, and E. Pagello. A Realisitic Simulation of a Humanoid Robot in USARSim. In *Proc. 4th Intl. Symp. on Mechatronics and its Applications (ISMA07)*, Sharjan, U.A.E., March 26-29 2007.

[73] D. Hähnel, W. Burgard, and G. Lakemeyer. GOLEX - Bridging the Gap between Logic (GOLOG) and a Real Robot. In *KI '98: Proc. of the 22nd Annual German Conference on Artificial Intelligence*, pages 165–176, London, UK, 1998. Springer-Verlag.

[74] P. Hamill. *Unit Test Frameworks*. O'Reilly, Sebastopol, CA, USA, 2005.

[75] T. Hemker, H. Sakamoto, M. Stelzer, and O. von Stryk. Efficient walking speed optimization of a humanoid robot. *International Journal of Robotics Research*, 28(2):303 – 314, Feb. 2009.

[76] R. Höpler. *A unifiying object-oriented methodology to consolidate multibody dynamics computations in robot control*. Fortschritt-berichte vdi, Technische Universität Darmstadt, Darmstadt, Germany, August 6 2004.

[77] R. Höpler and M. Thümmel. Symbolic Computation of the Inverse Dynamics of Elastic Joint Robots. In *Proc. of the 2004 IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 4314 – 4319, 03 2004.

[78] L. Hugues and N. Bredeche. Simbad: an Autonomous Robot Simulation Package for Education and Research. In *Proc. of the 2006 Intl. Conf. on the Simulation of Adaptive Behavior (SAB)*, Rome, Italy, 2006.

[79] J. Jackson. Microsoft Robotic Studio: A Technical Introduction. *Robotics and Automation Magazine*, 14(4):82–87, 2007.

[80] A. Jacoff, E. Messina, and J. Evans. Performance evaluation of autonomous mobile robots. *Industrial Robot: An International Jounal*, 29:259 – 267, 2002.

[81] R. E. Kalman. A New Approach to Linear Filtering and Prediction Problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[82] G. A. Kaminka, M. M. Veloso, S. Schaffer, C. Sollitto, R. Adobbati, A. N. Marshall, A. Scholer, and S. Tejada. GameBots: a flexible Test Bed for Multiagent Team Research. In *Commun. ACM*, volume 45, pages 43–45, New York, NY, USA, 2002. ACM.

[83] F. Kanehiro, H. Hirukawa, and S. Kajita. OpenHRP: Open Architecture Humanoid Robotics Platform. *The International Journal of Robotics Research*, 23(2):155–165, 2004.

[84] R. Kastner. Vergleich verschiedener Ansätze zur Koordinierung von Multi-Roboter Systemen. Diplomarbeit, Technische Universität Darmstadt, Department of Electrical Engineering, 2007.

[85] H. Kawata, A. Ohya, S. Yuta, W. Santosh, and T. Mori. Development of ultra-small lightweight optical range sensor system. In *Proc. of the 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2005.

[86] J. Kiener. *Heterogene Teams kooperierender autonomer Roboter (Heterogeneous Teams of Cooperating Robots)*. Fortschritt-berichte vdi, Technische Universität Darmstadt, December 15 2006.

[87] J. Kiener and O. von Stryk. Cooperation of heterogeneous, autonomous robots: A case study of humanoid and wheeled robots. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 959–964, San Diego, CA, USA, Oct. 29 - Nov. 2 2007.

[88] S. Klug, T. Lens, O. von Stryk, B. Möhl, and A. Karguth. Biologically Inspired Robot Manipulator for New Applications in Automation Engineering. In *Proceedings of Robotik 2008*, number 2012 in VDI-Berichte, Munich, Germany, June 11-12 2008. VDI Wissensforum GmbH.

[89] N. Koenig and A. Howard. Design and Use Paradigms of Gazebo, an Open-Source Multi-Robot Simulator. In *Proc. of the 2004 IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2004.

[90] S. Kohlbrecher. A Scalable, Plattform Independent SLAM System for Urban Search and Rescue. Diplomarbeit, Technische Universiät Darmstadt, Department of Computer Science, 2009.

[91] J. Kuffner, K. Nishiwaki, S.Kagami, M. Inaba, and H. Inoue. Online Footstep Planning for Humanoid Robots. In *Proc. of the 2003 IEEE Intl. Conf. on Robotics and Automation (ICRA)*. IEEE, September 2003.

[92] C. Kwok and D. Fox. Map-Based Multiple Model Tracking of a Moving Object. In D. Nardi, M. Riedmiller, C. Sammut, and J. Santos-Victor, editors, *RoboCup 2004: Robot Soccer World Cup VIII*, volume 3276 of *LNAI*. Springer, 2005.

[93] J.-C. Latombe. *Robot Motion Planing*. KAP, 1991.

[94] T. Laue and M. Hebbel. Automatic Parameter Optimization for a Dynamic Robot Simulation. In L. Iocchi, H. Matsubara, A. Weitzenfeld, and C. Zhou, editors, *RoboCup 2008: Robot Soccer World Cup XII*, number 5399 in LNAI, pages 121–132. Springer, 2009.

[95] T. Laue and T. Röfer. SimRobot - Development and Applications. In E. Menegatti, editor, *Workshop Proceedings of SIMPAR 2008, Intl. Conf. on Simulation, Modeling and Programming for Autonomous Robots*, pages 143–150, Venice, Italy, November 3-4 2008.

[96] T. Laue, K. Spiess, and T. Röfer. SimRobot - A General Physical Robot Simulator and Its Application in RoboCup. In A. Bredenfeld, A. Jacoff, I. Noda, and Y. Takahashi, editors, *RoboCup 2005: Robot Soccer World Cup IX*, number 4020 in Lecture Notes in AI, pages 173–183. Springer, 2006.

[97] H. J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.

[98] M. Lötzsch, M. Risler, and M. Jüngel. XABSL - A Pragmatic Approach to Behavior Engineering. In *Proc. of the 2006 IEEE/RSJ Intl. Conf. of Intelligent Robots and Systems (IROS)*, pages 5124–5129, Beijing, China, October 9-15 2006.

[99] D. G. Lowe. Object Recognition from Local Scale-Invariant Features. In *Proc. of the International Conference on Computer Vision*, pages 1150–1157, 1999.

[100] D. G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *Internation Journal of Computer Vision*, 60(2):91–110, 2004.

[101] M. J. Matarić and F. Michaud. *Springer Handbook of Robotics*, chapter Behavior-Based-Systems, pages 891–909. Springer, 2008.

[102] R. B. McGhee and A. A. Frank. On the Stability Properties of Quadruped Creeping Gaits. *Mathematical Biosciences*, 3(1-2):331 – 351, 1968.

[103] P. J. McKerrow. *Introduction to Robotics*. Addison-Wesley, 1991.

[104] E. Menegatti, G. Silvestri, E. Pagello, N. Greggio, A. Cisternino, F. Mazzanti, R. Sorbello, and A. Chella. 3D Models of Humanoid Soccer Robot in USARSim and Robotics Studio Simulators. *International Journal of Humanoid Robotics*, 5(3):523–546, 2008.

[105] E. Messina and A. Jacoff. Performance Standards for Urban Search and Rescue Robots. In *Proc. of the SPIE Defense and Security Symposium*, 2006.

[106] O. Michel. Cyberbotics Ltd. - Webots(TM): Professional Mobile Robot Simulation. *Intl. Journal of Advanced Robotic Systems*, 1(1):39–42, 2004.

[107] I. Millington. *Game physics engine development*. Morgan Kaufmann Publishers, 2007.

[108] S. Morgan. *Programming Microsoft Robotics Studio*. Microsoft Press, 2008.

[109] C. Niehaus, T. Röfer, and T. Laue. Gait Optimization on a Humanoid Robot using Particle Swarm Optimization. In *Proc. of the 2nd Workshop on Humanoid Soccer Robots at the 2007 IEEE-RAS Intl. Conf. on Humanoid Robots*, Pittsburgh, PA, USA, Nov. 29 - Dec. 1 2007.

[110] O. Obst and M. Rollmann. SPARK – A Generic Simulator for Physical Multiagent Simulations. *Computer Systems Science and Engineering*, 20(5):347–356, Sept. 2005.

[111] H. Olsson. *Control Systems with Friction*. PhD thesis, Dept. of Automatic Control, Lund Inst. of Technology, 1996.

[112] F. Otsuka, H. Fujii, and K. Yoshida. Development of Three Dimensional Dynamics Simulator with Omnidirectional Vision Model. In G. Lakemeyer, E. Sklar, D. Sorrenti, and T. Takahashi, editors, *RoboCup 2006: Robot Soccer World Cup XI*, volume 4434 of *LNAI*. Springer, 2007.

[113] D. Pachur, T. Laue, and T. Röfer. Real-time Simulation of Motion-based Camera Disturbances. In L. Iocchi, H. Matsubara, A. Weitzenfeld, and C. Zhou, editors, *RoboCup 2008: Robot Soccer World Cup XII*, volume 5399 of *LNAI*, pages 591–601, 2009.

[114] C. Pepper, S. Balakirsky, and C. Scrapper. Robot Simulation Physics Validation. In *Proc. of the Performace Metrics for Intelligent Systems (PERMIS) Workshop 2007*, pages 97–104, Galthersburgh, USA, 2007.

[115] K. Petersen. Effiziente Kollisionserkennung und echtzeitfähige Simulation der Kinematik, Dynamik und Sensorik autonomer Fahrzeuge. Diplomarbeit, Technische Universität Darmstadt, Department of Computer Science & Department of Mathematics, 2007.

[116] S. Petters and D. Thomas. RoboFrame - Softwareframework für mobile autonome Robotersysteme. Diplomarbeit, Technische Universität Darmstadt, Department of Computer Science, 2005.

[117] S. Petters, D. Thomas, M. Friedmann, and O. von Stryk. Multilevel Testing of Control Software for Teams of Autonomous Mobile Robots. In S. C. et al., editor, *Simulation, Modeling and Programming for Autonomous Robots (SIMPAR 2008)*, number 5325 in LNAI, pages 183–194. Springer, Nov. 4-6 2008.

[118] S. Petters, D. Thomas, and O. von Stryk. RoboFrame - A Modular Software Framework for Lightweight Autonomous Robots. In *Proc. of the Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware at the 2007 IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, San Diego, CA, USA, Oct. 29 2007.

[119] M. Prager. Untersuchung von Dynamikalgorithmen zur Echtzeitsimulation von Humanoidrobotern. Bachelorsthesis, Technische Universität Darmstadt, Department of Computer Science, 2009.

[120] M. Risler. *Behavior Control for Single and Multiple Autonomous Agents Based on Hierarchical Finite State Machines*. PhD thesis, Technische Universität Darmstadt, May 15 2009.

[121] M. Risler and O. von Stryk. Formal behavior specification of multi-robot systems using hierarchical state machines in XABSL. In *Proc. of the AAMAS08-Workshop on Formal Models and Methods for Multi-Robot Systems*, Estoril, Portugal, May 12-16 2008.

[122] Robotis. User Manual: RX-28. Technical report, Robotis, 2007.

[123] Robotis. User Manual: RX-64. Technical report, Robotis, 2007.

[124] T. Röfer, T. Laue, A. Burchardt, E. Damrose, K. Gillmann, C. Graf, T. J. de Haas, A. Härtl, A. Rieskamp, A. Schreck, and J.-H. Worch. B-Human Team Report and Code Release 2008. Technical report, 2008.

[125] T. Röfer, T. Laue, M. Weber, H.-D. Burkhard, M. Jüngel, D. Göhring, J. Hoffmann, B. Altmeyer, T. Krause, M. Spranger, O. von Stryk, R. Brunn, M. Dassler, M. Kunz, T. Oberlies, M. Risler, U. Schwiegelshohn, M. Hebbel, W. Nistico, S. Czarnetzki, T. Kerkhof, M. Meyer, C. Rohde, B. Schmitz, M. Wachter, T. Wegner, and C. Zarges. GermanTeam 2005. Technical report, HU-Berlin, U-Bremen, TU-Darmstadt, U-Dortmund, 2005.

[126] R. G. Sargent. Verification and Validation of Simulation Models. In *Proc. of the 2003 Winter Simulation Conference*, 2003.

[127] T. Schmits and A. Visser. An Omnidirectional Camera Simulation for the USARSim World. In L. Iocchi, H. Matsubara, A. Weitzenfeld, and C. Zhou, editors, *RoboCup 2008: Robot Soccer World Cup XII*, volume 5399 of *LNAI*. Springer, 2009.

[128] M. Schobbe and P. Stamm. Modulare Weltmodellierung und Kommunikation in heterogenen Robotersystemen. Diplomarbeit, Technische Universität Darmstadt, Department of Computer Science, 2007.

[129] D. Scholz. Modulare und plattformunabhängige Bewegungserzeugung für autonome mobile humanoide Roboter. Diplomarbeit, Technische Universität Darmstadt, Department of Computer Science, 2008.

[130] C. Scrapper, S. Balakirsky, and E. Messina. MOAST and USARSim: a Combined Framework for the Development and Testing of Autonomous Systems. In *Proc. of the SPIE Defense and Security Symposium*, 2006.

[131] M. Segal and K. Akeley. The OpenGL Graphics System: A Specification (Version 2.0 - October 22, 2004). Technical report, Silicon Graphics, Inc., 2004.

[132] P. Shirley, M. Ashikhmin, M. Gleicher, S. Marschner, E. Reinhard, W. Sung, K. Thompson, and P. Willemsen. *Fundamentals of Computer Graphics*. A K Peters, 2005.

[133] D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide, Sixth Edition, The Official Guide to Learning OpenGL, Version 2.1*. Addison-Wesley Professional, 2007.

[134] R. Smith. Open Dynamics Engine v0.5 User Guide. Technical report, 2006.

[135] R. Smith. ODE - Open Dynamics Engine, http://www.ode.org, 2007.

[136] S. Smolorz. Echtzeit-Linux mit Xenomai. *Elektronik*, 3:86 – 90, 2007.

[137] M. Stelzer. *Forward Dynamics Simulation and Optimization of Walking Robots and Humans*. Fortschritt-berichte vdi, Technische Universität Darmstadt, May 24 2007.

[138] M. Stelzer, M. Hardt, and O. von Stryk. Efficient Dynamic Modeling, Numerical Optimal Control and Experimental Results for Various Gaits of a Quadruped Robot. In *CLAWAR 2003: 6th Intl. Conf. on Climbing and Walking Robots, Catania, Italy*, pages 601–608, Sept. 17-19 2003.

[139] G. Stoll. A supporter behavior for autonomous soccer playing humanoid robots. Bachelorsthesis, Technische Universität Darmstadt, 2009.

[140] S. Thrun, M. Bennewitz, W. Burgard, A. B. Cremers, F. Dellaert, D. Fox, D. Hähnel, C. R. Rosenberg, N. R., J. Schulte, and D. Schulz. MINERVA: A Second-Generation Museum Tour-Guide Robot. In *Proc. of the 1999 IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 1999–2005, 1999.

[141] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. The MIT Press, September 2005.

[142] S. Thrun, D. Fox, W. Burgard, and F. Dellaert. Robust Monte Carlo Localization for Mobile Robots. *Journal of Artificial Intelligence Research*, 128:99 – 141, 2001.

[143] A. Vatcheva. Sensorische und motorische Basisfähigkeiten für ein autonomes Bodenfahrzeug. Diplomarbeit, Technische Universität Darmstadt, Department of Computer Science, 2009.

[144] R. Vaughan. Massively multi-robot simulation in stage. *Swarm Intelligence*, 2:189–208, 2008.

[145] M. Vukobratović and B. Borovac. Zero Moment Point – Thirty Five Years of its Life. *Internation Journal of Humanoid Robotics*, 1(1):157–173, 2004.

[146] M. W. Walker and D. E. Orin. Efficient Dynamics Computer Simulation of Robotic Mechanisms. *Journal of Dynamic Systems, Measurement, and Control*, 104:205–211, 1982.

[147] J. Wang and S. Balakirsky. UARSSim V3.1.3 - A Game-based Simulation of mobile robots. Technical report, NIST, 2008.

[148] J. C. Zagal and J. R. del Solar. Learning to Kick the Ball Using Back to Reality. In D. Nardi, M. Riedmiller, C. Sammut, and J. Santos-Victor, editors, *RoboCup 2004: Robot Soccer World Cup VIII*, volume 3276 of *LNAI*. Springer, 2005.

[149] J. C. Zagal and J. R. del Solar. UCHILSIM: A dynamically and visually realistic simulator for the robocup four legged league. In D. Nardi, M. Riedmiller, C. Sammut, and J. Santos-Victor, editors, *RoboCup 2004: Robot Soccer World Cup VIII*, volume 3276 of *LNAI*. Springer, 2005.

[150] J. C. Zagal, J. Ruiz-del Solar, and V. P. Back to Reality: Crossing the Reality Gap in Evolutionary Robotics. In *Proc. of the 5th IFAC/EURON Symposium on Intelligent Autonomous Vehicles*, 2004.

[151] J. C. Zagal, I. Sarmiento, and J. Ruiz-del Solar. An Application Interface for UCHILSIM and the arrival of new challenges. In A. Bredenfeld, A. Jacoff, I. Noda, and Y. Takahashi, editors, *RoboCup 2005: Robot Soccer World Cup IX*, volume 4020 of *LNAI*, pages 464–471. Springer, 2006.

[152] M. Zaratti, M. Fratarcanglei, and L. Iocchi. A 3D Simulator of Multiple Robots based on UARSSim. In G. Lakemeyer, E. Sklar, D. Sorrenti, and T. Takahashi, editors, *RoboCup 2006: Robot Soccer World Cup XI*, volume 4434 of *LNAI*. Springer, 2006.

[153] V. A. Ziparo, L. Iocchi, D. Nardi, P. F. Palamara, and H. Costelha. Petri Net Plans: a Formal Model for Representation and Execution of Multi-Robot Plans. In *AAMAS '08: Proc. of the 7th Intl. joint Conf. on Autonomous Agents and Multiagent Systems*, pages 79–86, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

Wissenschaftlicher Werdegang[1]

06/1994	Allgemeine Hochschulreife
10/1995 - 09/1998	Studium der Informationstechnik an der Berufsakademie Mannheim
09/1998	Dipl. Ing. (BA) Informationstechnik
10/1998 - 09/2003	Studium der Informatik an der Technischen Universität Darmstadt
09/2003	Diplom in Informatik
seit 01/2004	Doktorand am Fachbereich Informatik, Technische Universität Darmstadt
05/2004 - 06/2005	Hilfskraft mit Abschluss, Fachbereich Informatik, Technische Universität Darmstadt
seit 07/2005	Wissenschaftlicher Mitarbeiter, Fachbereich Informatik, Technische Universität Darmstadt

Erklärung[2]

Hiermit erkläre ich, dass ich die vorliegende Arbeit - mit Ausnahme der ausdrücklich genannten Hilfsmittel - selbständig verfasst habe.

[1] gemäß § 20 Abs. 3 der Promotionsordnung der TU Darmstadt
[2] gemäß § 9 Abs. 1 der Promotionsordnung der TU Darmstadt